Contents

OBJECT ORIENTED PROGRAMMING & METHODOLOGY
(III-SEM., CS-BRANCH)

PAGE NO.

UNIT-I :

Introduction to Object Oriented Thinking & Object Oriented Programming : Comparison with Procedural Programming, features of Object oriented paradigm – Merits and demerits of OO methodology; Object model, Elements of OOPS, IO processing ....(03 to 38)

UNIT-II :

Encapsulation and Data Abstraction, Concept of objects – State, Behaviour & Identity of an object............(39 to 52)

Classes – Identifying classes and candidates for Classes Attributes and Services, Access modifiers, Static members of a Class, Instances, Message passing, and Construction and destruction of Objects ....(52 to 94)

UNIT-III :

Relationships – Inheritance – Purpose and its types, 'is a' relationship ....(95 to 128)

Association, Aggregation ....(128 to 142)

Concept of interfaces and Abstract classes ....(142 to 156)

UNIT-IV :

Polymorphism – Introduction, Method Overriding & Overloading, static and run time Polymorphism ....(157 to 192)

UNIT-V :

Strings ....(193 to 202)

Exceptional handling ....(202 to 219)

Introduction of Multi-threading and Data collections ....(219 to 249)

Case study like – ATM, Library management system............(249 to 256)

MSInvelter

UNIT 1

INTRODUCTION TO OBJECT ORIENTED THINKING AND OBJECT ORIENTED PROGRAMMING – COMPARISON WITH PROCEDURAL PROGRAMMING, FEATURES OF OBJECT ORIENTED PARADIGM, MERITS AND DEMERITS OF OO METHODOLOGY, OBJECT MODEL, ELEMENTS OF OOPS, I/O PROCESSING

Q.1. Explain in brief about object oriented (OO) thinking.

Ans. An object-oriented programming was developed because limitations were discovered in the earlier approaches of programming.

In the procedural languages, like C, Pascal, FORTRAN and others, the statement tells the computer to do something, that is, get some input and after manipulation display that output. Here, programs are a list of instructions. For very small programs, there is no need to organize principle. But when programs become larger, a single list of instructions becomes unwieldy. For this, function was adopted to make programs more comprehensible. A procedural program is divided into functions and each function has a clearly defined purpose. The breaking idea of a program into functions can be extended by grouping a number of functions together into a larger entity called a ***module***.

As programs grow ever larger and more complex, even the structured programming approach begins to show signs and strain. The project is too complex, the schedule slips, complexity increases, the schedule slips further and disaster ensures. Analyzing the reasons for these failures reveals that there are weaknesses in the procedural paradigm. No matter how well the structured programming approach is implemented, large programs become excessively complex.

There are two related problems –

(i) Functions have unrestricted access to global data.

(ii) Unrelated functions and data, the basis of the procedural paradigm, provide a poor model of the real world.

The fundamental idea behind object-oriented languages is to combine into a single unit both data and functions that operate on that data. Such a unit is called an object. In this sense, we cannot access data directly. The data is hidden, so it is safe from accidental alteration.

Therefore, we can say that an object-oriented programming approach can solve the problem which occur in the procedural-oriented programming approach.

The term *object-oriented* (OO) means that we organize software as a collection of discrete objects that incorporate both data structure and behaviour. This is in contrast to conventional programming in which data structure and behaviour are only loosely connected. There is some dispute about exactly what characteristics are required by an object-oriented approach, but they generally include four aspects – identity, classification, polymorphism, and inheritance.

Identity means that data is quantized into discrete, distinguishable entities called objects. A *paragraph in a document*, a *window on my workstation*, and the white queen in a chess game are examples of objects. Fig. 1.1 shows some additional objects. Objects can be concrete, such as a *file* in a file system, or conceptual, such as a *scheduling policy* in a multiprocessing operating system. Each object has its own inherent identity. In other words, two objects are distinct even if all their attribute values are identical.

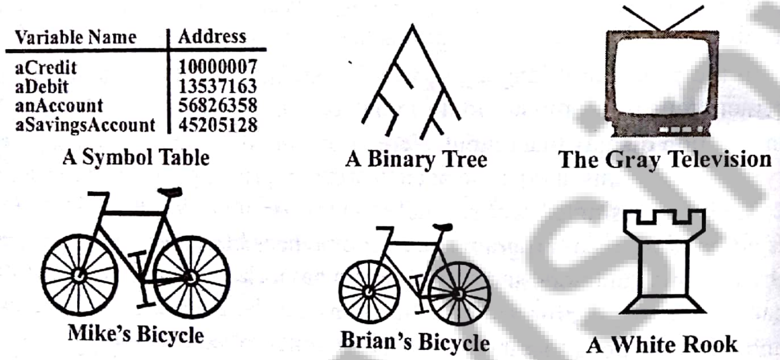


Fig. 1.1 Objects

Classification means that objects with the same data structure (attributes) and behaviour (operations) are grouped into a class. *Paragraph*, *Monitor*, and *ChessPiece* are examples of classes. A class is an abstraction that describes properties important to an application and ignores the rest. Any choice of classes is arbitrary and depends on application.

Each class describes a possibly infinite set of individual objects. Each object is said to be an *instance* of its class. Each instance of the class has its own value for each attribute but shares the attribute names and operations with other instances of the class. Fig. 1.2 shows two classes and some of their respective instance objects. An object contains an implicit reference to its own class; "it knows what kind of thing it is".

Polymorphism means that the same operation may behave differently on different classes. The *move* operation, for example, behaves differently for a Pawn than for the queen in a chess game. An *operation* is an action or transformation that an object performs or is subject to. *Right-justify*, *display*, and move are examples of operations. A specific implementation of an operation by a certain class is called a *method*. Because an object-oriented operator is polymorphic, it may have more than one method implementing it.

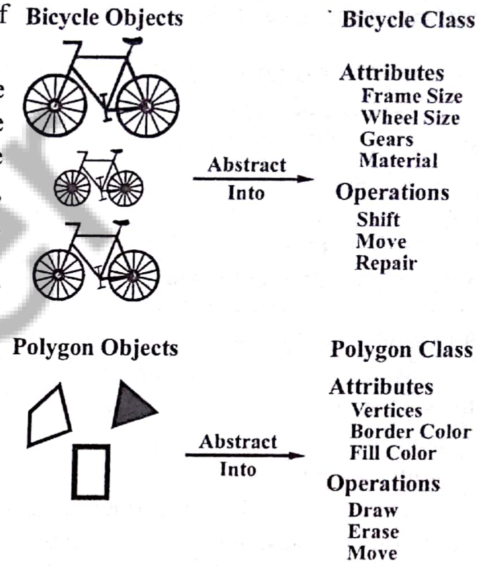


Fig. 1.2 Objects and Classes

In the real world, an operation is simply an abstraction of analogous behaviour across different kinds of objects. Each object "knows how" to perform its own operations. In an object-oriented programming language, however, the language automatically selects the correct method to implement an operation based on the name of the operation and the class of the object being operated on. The user of an operation need not be aware of how many methods exist to implement a given polymorphic operation. Developers can add new classes without changing existing code, as long as they provide methods for each applicable operation.

Inheritance is the sharing of attributes and operations (features) among classes based on a hierarchical relationship. A superclass has general information that subclasses refine and elaborate. Each subclass incorporates, or inherits, all the features of its superclass and its own unique features. Subclasses need not repeat the features of the superclass. For example, *ScrollingWindow* and *FixedWindow* are subclasses of *Window*. Both subclasses inherit the features of *Window*, such as a visible region on the screen. *ScrollingWindow* adds a scroll bar and an offset. The ability to factor out common features of several classes into a common superclass and to inherit the features from the superclass can greatly reduce repetition within designs and programs and is one of the main advantages of an object-oriented technology.

Q.2. Write the need of object oriented programming. **(R.G.P.V., Dec. 2014)**

Ans. Refer to Q.1.

Q.3. What is meant by object-oriented analysis ? What are its benefits ?

Ans. The definition of an object-oriented analysis model encompasses a description of the static and dynamic characteristics of classes that describe a system or product. This activity is performed by a software engineer. Before we can build an object-oriented system, we have to define the classes that represent the problem to be solved, the manner in which the classes relate to and interact with one another, the inner workings of objects and the communication mechanisms that allow them to work together. All of these things are accomplished during object-oriented analysis.

We cannot build software until we have a reasonable understanding of the system or product. OOA provides us with a concrete way to represent our understanding of requirements and then test that understanding against the customer's perception of the system to be built.

Object-oriented analysis (OOA) begins with a description of use cases – a scenario-based description of how actors (people, machines, and other systems) interact with the product to be built. Class responsibilities-collaborator (CRC) modeling translates the information contained in use-cases into a representation of classes and their collaborations with other classes. The static and dynamic characteristics of classes are then modeled using a united modeling language.

Object-oriented analysis can yield following benefits –

(i) Maintainability – Maintainability through simplified mapping to the real world, which provides for less analysis effort, less complexity in system design, and easier verification by the user.

(ii) Reusability – Reusability of the analysis artifacts which saves time and costs and depending on analysis method and programming language.

(iii) Productivity – Productivity gains through direct mapping to features of object-oriented programming language.

Q.4. What is meant by object-oriented design ?

Ans. The design of object-oriented software requires the definition of a multilayered software architecture, the specification of subsystems that perform required functions and provide infrastructure support, a description of objects that form the building blocks of the system, and a description of the communication mechanisms that allow data to flow between layers, subsystems and objects. Object-oriented design accomplishes all of these things. Object-oriented design is performed by a software engineer.

An object-oriented system draws upon class definitions that are derived from the analysis model. Some of these definitions will have to be built from scratch, but many others may be reused, if appropriate design patterns are

recognized. Object-oriented design establishes a design blueprint that enables a software engineer to define the object-oriented architecture in a manner that maximizes reuse, thereby, improving development speed and end-product quality.

Object-oriented design is divided into two major activities – system design and object design. System design creates the product architecture defining a series of "layers" that accomplishes specific system functions and identifying the classes that are encapsulated by subsystems that reside at each layer. In addition, system design considers that specification of three components, the user interface, data management functions, and task management facilities. Object design focuses on the internal details of individual classes, defining attributes operations, and message details.

An object-oriented design model encompasses software architecture, user interface description, data management components, task management facilities, and detailed descriptions of each class to be used in the system. At each stage, the elements of object-oriented design model are reviewed for clarity, correctness, completeness, and consistency with customers requirements and with one another.

Q.5. What is object oriented programming (OOP) ? How is it near to real world ? **(R.G.P.V., Dec. 2013)**

Ans. Object-oriented programming is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand. Thus, an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

Object-oriented programming is a practical and useful programming methodology that encourages modular design and software reuse. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions. OOP allows decomposition of problem into a number of entities called objects and then builds data and functions around these objects.

The basic philosophy of using objects in programs is a simple one – that the world is composed of interacting, classifiable and identifiable objects, and therefore programs, too, can usefully be structured in this way. Since all software applications serve people living in the real world, then that software should mirror the real world as closely as possible, both in its working and in its underlying design. Simply, object-oriented programming is a more 'natural' way to program.

Q.6. How does object-oriented approach differ from object-base approach ?

Ans. The major differences between object-based and object-oriented approaches are as follows –

S.No.	*Object-based Approach*	*Object-oriented Approach*
(i)	The major features of object-based programming are – data encapsulation, data hiding, automatic initialization, clear-up of objects and operator overloading.	It includes all the features supported by object-based approach along with two additional features of inheritance and dynamic binding.
(ii)	Languages that support object-based programming are called object-based programming languages. They do not support inheritance and dynamic binding.	Languages that support object-oriented approach has all facilities of encapsulation, inheritance, polymorphisms, and dynamic binding, and hence, are known as object-oriented languages.
(iii)	It supports abstract data but not classes.	It supports both abstract data and classes which provide inheritance and polymorphism.
(iv)	The languages that support object-based programming are 83 Ada and Modula-2.	The languages that support object-oriented approaches are Ada 95, Modula-3, C++, Smalltalk, Java, etc.

Q.7. What are promising areas for application of object-oriented programming ?

Ans. The most popular application of object-oriented programming has been area of user interface design such as windows. Hundreds of windowing system have been developed using the object-oriented programming techniques. The promising areas for application of object-oriented programming include –

(i) Real time system
(ii) AI and expert systems
(iii) Simulation and modeling
(iv) Object-oriented databases
(v) CAD/CAM systems
(vi) Office automation
(vii) Neural networks and parallel programming
(viii) Expertext, hypertext and hypermedia.

Real-business systems are often much more complex and contain many more objects with complicated attributes and methods. Object-oriented programming is useful in these types of applications because it can simplify complex problem. The object-oriented paradigm sprang from the language has matured into design, and has recently moved into analysis.

Application of object-oriented programming are beginning to gain importance in many areas. The richness of object-oriented programming environment will enable the software industry to improve not only the quality of software systems but also its productivity. Object-oriented technology is certainly going to change the way the software engineers think, analyze, design and implement future systems.

Q.8. What are the major characteristics of object-oriented langauages ? Explain. ***(R.G.P.V., Dec. 2013)***

Or

Name the characteristics of object-oriented programming. ***(R.G.P.V., Dec. 2015)***

Or

What are the main characteristics of object-oriented language ? ***(R.G.P.V., Nov. 2018)***

Ans. The basic characteristics of object-oriented programming are as follows –

(i) Emphasis is on data rather than procedure.
(ii) Programs are divided into what are known as objects.
(iii) Data structures are designed such that they characterize the objects.
(iv) Functions that operate on the data of an object are tied together in the data structure.
(v) Data is hidden and cannot be accessed by external functions.
(vi) Objects may communicate with each other through functions.
(vii) New data and functions can be easily added whenever necessary.
(viii) Follows bottom-up approach in program design.

Q.9. What are the various features supported by object-oriented programming languages ? Explain each of them.

Ans. Some of the concepts (or features) used extensively in object-oriented programming are as follows –

(i) Objects – Objects are the basic run-time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represent user-defined data such as vectors, time and lists. Programming problem is analyzed in terms of objects and the nature of communication between them.

When a program is executed, the objects interact by sending messages to one another. For example, if customer and account are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance. Each object contains data, and code to manipulate the data. Objects can interact without having to know details of each other's data or code.

(ii) Classes – A class is a collection of objects of similar type. Obje are variables of the type class. Once a class has been defined, we can crea any number of objects belonging to that class. Each object is associated wi the data of type class with which they are created. Classes are user-defin data types. For example, mango, apple and orange are members of the cla fruit, if fruit has been defined as a class, then the statement –

Fruit mango;

where, mango is an object.

(iii) Data Abstraction and Encapsulation – The wrapping up data and functions into a single unit (called class) is known as encapsulatio Data encapsulation is the most striking features of a class. The data is n accessible to the outside world, and only those functions which are wrapp in the class can access it. This insulation of data from direct access by th program is called data hiding or information hiding.

Abstraction refers to the act of representing essential features of an objec without including the background details or explanations. Classes use th concept of abstraction and are defined as a list of abstract attributes such a size, weight and cost, and function to operate on these attributes. The encapsulate all the essential properties of the objects that are to be created The attributes are sometimes called data members, because they hol information. The functions that operate on these data are sometimes calle methods or member functions. Since the classes use the concept of dat abstraction, they are known as Abstract Data Types (ADT).

(iv) Inheritance – Inheritance is the process by which objects o one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. The concept of inheritance provides the idea of reusability. This means that we can add additional features to an existin class without modifying it.

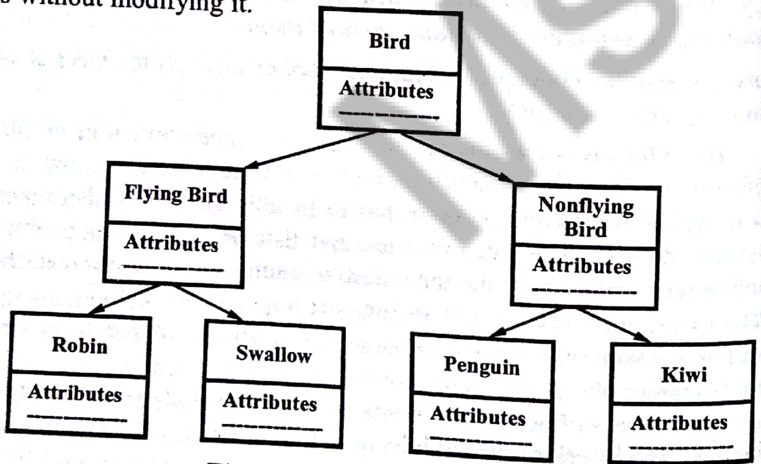


Fig. 1.3 Property Inheritance

For example, the bird 'robin' is a part of the class 'flying bird' which is again a part of the class 'bird'.

(v) Polymorphism – It means the ability to take more than one form i.e., 'one name, multiple forms'. In OOP an operation may exhibit different behaviours in different instances. Concept of polymorphism is implemented using the overloaded functions and operators. The overloaded member functions are 'selected' for invoking by matching arguments, both type and number.

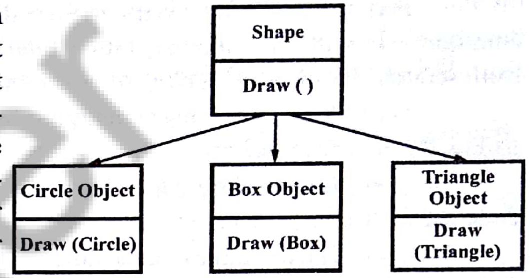


Fig. 1.4 Polymorphism

(vi) Dynamic Binding – Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding (also known as late binding) means that the code associated with a given procedure call is not known until the time of the call at run-time.

(vii) Message Passing – An object-oriented program consists of a set of objects that communicate with each other. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

Message passing involves specifying the name of the object, the name of the function (message) and the information to be sent.

(viii) Reusability – Once a class has been written, created and debugged, it can be distributed to other programs for use, this is called reusability. In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is done by deriving a new class from the existing one. The new class will inherit the capabilities of the old one, but is free to add new features of its own.

Q.10. What are the benefits and risks of object-oriented development ?

Or

Write merits and demerits of object-oriented programming languages as compared to procedure-oriented languages.

Ans. Object-oriented development is a new way of thinking about software based on abstractions that exist in the real world. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The major benefits of object-oriented development are as follows –

(i) Object-oriented systems can be easily upgraded from small to large system.

(ii) The data centered design approach enables us to capture more details of a model in an implementable form.

(iii) It is easy to partition the work in a project based on objects.

(iv) It is possible to map objects in problems domain to those in the progra

(v) It is possible to have multiple instances of an object to co-exi without any interference.

(vi) We can build programs from the standard working modules th communicate with one another, rather than having to start writing the cod from scratch. This leads to saving of development time and higher productivi

(vii) Message passing techniques for communication between objec makes the interface descriptions with external systems much simpler.

(viii) We can eliminate redundant code and extend the use of existin code through inheritance.

(ix) The principle of data hiding helps the programmer to build secu programs that cannot be invaded by code in other parts of the program.

(x) Software complexity can be easily managed.

(xi) We can categorize the object into various classes.

Other advantages are reuse, an emphasis on modeling the real world, consistent and seamless package, naturalness, resistance to change encapsulation and abstraction, etc.

Though it is possible to incorporate all these features in an object-oriente system, their importance depends on the type of the project and the preferenc of the programmer.

The specific risks involved in applying object-oriented approach to software problem include –

(i) Resource Demands – Since an object-oriented program can requir a much greater processing overhead than one written using traditional methods, i may work more slowly. This is not good from a customer point of view.

(ii) Object-persistance – An object's natural environment is in RAM as a dynamic entity. This is in contrast to traditional data storage in files o databases where the natural environment of the data is on external storage This causes problem when we want objects to persist between runs of program, even more so between different application.

(iii) Complexity – The message passing between many objects in complex application can be difficult to trace and debug.

(iv) Reusability – When inheritance is used it is not easy to produce reusable objects between applications, since it makes their classes closely coupled to the rest of the hierarchy. Object can become application specific to reuse with inheritance. It is extremely difficult to coordinate very large systems

Q.11. Write merits and demerits of object oriented methodology.

(R.G.P.V., Nov. 2018)

Ans. Refer to Q.10.

Q.12. Write the features of procedure-oriented, object-based and object-oriented languages separately. Also write merits and demerits of each.

Ans. **Procedure-oriented Languages** – C, Pascal, FORTRAN, and similar languages are procedural languages. That is each statement in the language tells the computer to do something. A program in a procedural language is a list of instructions. Features of procedure-oriented languages are as follows –

(i) Emphasis is on doing things.
(ii) Large programs are divided into smaller programs known as functions.
(iii) Most of functions share global data.
(iv) Data move openly around the system from function to function.
(v) Functions transform data from one form to another.
(vi) Employs top-down approach in program design.

Merit – Procedure-oriented programming basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups. We can use a flowchart to organize these actions and represent the flow of control from one action to another.

Demerit – In a large program it is very difficult to identify what data is used by which function. Another serious drawback with the procedural approach is that it does not model real world problems very well. This is because functions are action-oriented and do not really correspond to the elements of the problem.

Object-based Languages – It is the style of programming that primarily supports encapsulation and object identity. Major features that are required for object based programming are as follows –

(i) Data encapsulation
(ii) Data hiding and access mechanisms
(iii) Automatic initialization and clear-up of objects
(iv) Operator overloading.

Languages that support programming with objects are said to be object-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based language.

Object-oriented Languages – It incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding. Object-oriented language can therefore be characterized by the following statements –

Object-based features + Inheritance + Dynamic binding

Merit and Demerit – Refer to Q.10.

Q.13. Describe how data are shared by functions in a procedure-oriented program.

Ans. Procedure-oriented programming basically consists of writing a list of instructions for the computer to follow, and organizing these instructions

into groups known as functions. We use a flowchart to organize these actio and represent the flow of control from one action to another. While w concentrate on the development of functions, very little attention is given the data that are being used by various functions. Fig. 1.5 shows the relati between data and functions in a procedure-oriented program.

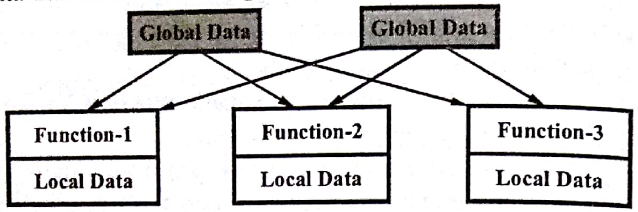


Fig. 1.5 Relationship of Data and Functions in Procedural Programmin

Global data are more vulnerable to an inadvertent change by a functio In a large program it is very difficult to identify what data is used by whic function. In case, we require to revise an external data structure, we also nee to revise all functions that access the data. This creates an opportunity fo bugs to creep in. Another drawback of this approach is that it does not mode real world problems very well. This is because functions are action-oriente and do not really correspond to the elements of the problem.

Q.14. How are data and functions organized in an object oriente program ?

Ans. Object oriented programming (OOP) treats data as a critical element in the program development and does not permit it to flow freely around the system. It ties data closely to the functions that operate on it and prevents it from modification from outside functions. OOP decompose a problem into a number of entities known as objects and then builds data and function near these objects. The data of an object can be accessed only by its functions. However, functions of one object can access the functions of other objects. Fig. 1.6 shows the organization of data and functions in OOP.

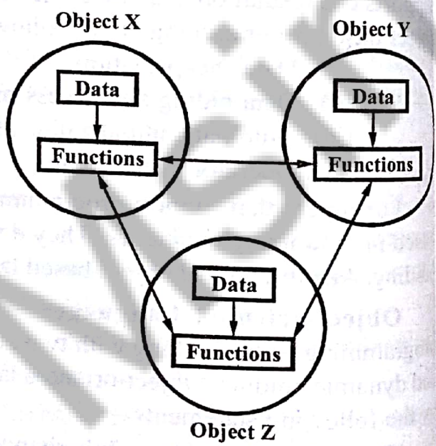


Fig. 1.6

Q.15. What are fundamental differences between object oriented programming and procedure/structured oriented programming ?

Ans. The major differences between OOP and procedure/stryctyred-oriented programming are as follows –

S. No.	Object-oriented Programming	Procedure-oriented Programming
(i)	It emphasizes on data rather than procedure.	It emphasizes on doing things or algorithms.
(ii)	Programs are decomposed into objects.	Large programs are decomposed into functions.
(iii)	Data structures used characterize the objects.	Most of the functions share global data.
(iv)	Data is hidden and cannot be accessed by external functions.	Data move openly around the system from function to function.
(v)	OOP follows bottom-up approach in program design.	POP follows top-down approach in program design.

Q.16. What is object oriented programming ? How it is different from procedure oriented programming ? **(R.G.P.V., June 2015)**

Ans. Refer to Q.5 and Q.15.

Q.17. What is object oriented programing ? How is it different from procedure oriented programming ? What the the unique advantages are of object oriented program ? **(R.G.P.V., June 2013)**

Or

What is object oriented programming ? How does it differ from structured programming ? Discuss the characteristics of object oriented languages. **(R.G.P.V., June 2016)**

Ans. Refer to Q.5, Q.15 and Q.8.

Q.18. What are the different types of models of object modeling technique ? Explain briefly.

Ans. The three types of models of object modeling technique are as follows –

(i) Object Model – Object model describes the structure of objects in a system – their identity, their relationships to other objects, their operations and their attributes. It provides the essential framework into which the ***dynamic and functional*** models can be placed. We can say that objects are building blocks of model.

The goal in constructing an object model is to capture those concepts from the real world that are important to an application. In modeling an engineering problem, the object model should contain terms familiar to engineers; in modeling a business problem, terms from the business; in modeling a user interface, terms from the application domain. An analysis model should not contain computer constructs unless the application being modeled is inherently a computer problem, such as a compiler or an operating system. The design model describes how to solve a problem and may contain computer constructs.

This model is graphically represented with object diagrams containing object classes. Classes are arranged into hierarchies sharing common structure and behaviour and are associated with other classes. Classes define the attribute values carried by each object instance and the operations which each object performs or undergoes.

(ii) Dynamic Model – The dynamic model describes those aspects of a system concerned with time and the sequencing of operations such as events that mark changes, sequences of events, states that define the context for events, and the organization of events and states. The dynamic model captures ***control***, that aspect of a system that describes the sequences of operations that occur, without regard for what the operations do, what they operate on, or how they are implemented.

The dynamic model is represented graphically with state diagrams. Each state diagram shows the state and event sequences permitted in a system for one class of objects. State diagrams also refer to the other models. Actions in the state diagrams correspond to functions from the functional model. Events in a state diagram become operations on objects in the object model.

(iii) Functional Model – A functional model is a description of aspects of a system that transform values using functions, mappings, constraints and functional dependencies.

The functional models show how output values in a computation are derived from input values without regard for the order in which the values are computed. It consists of multiple data flow diagram which shows the flow of values from external inputs, through operation and internal data stores, to external output.

It also shows a computation and the functional derivation of the data values in it without indicating how, when, or why the values are computed. The dynamic model controls which operations are performed and the order in which they are applied. The object model defines the structure of value that the operation operates on. For batch-like computations such as compiler or numerical computations, the functional model is the primary model, but in large system all three models are important.

In functional modeling the ***data flow diagram*** shows the relationship between values in a computation. A data flow diagram is a graph of process. Data flow diagram can be nested hierarchically, but ultimately the leaf processes must be specified directly as operations.

Q.19. What are the different elements of object model ?

(R.G.P.V., June 2010, 2016)

Ans. There are four major elements of the object model –

(i) Abstraction (ii) Encapsulation
(iii) Modularity (iv) Hierarchy.

By major, we mean that a model without any one of these elements is not object-oriented.

There are three minor elements of object model –

(i) Typing (ii) Concurrency
(iii) Persistence.

By minor, we mean that each of these elements is a useful, but not essential part of the object model.

Abstraction – An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.

An abstraction is one of the fundamental ways that we as human cope with complexity. An abstraction focuses on the outside view of an object, and so serves to separate an object's essential behaviour from its implementation.

Hoare suggests, "abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon these similarities and to ignore for the time being the differences."

Shaw defines, "a simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others. A good abstraction is one that emphasizes details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary."

There are the following types of abstraction –

(i) Entity Abstraction – An object that represents a useful model of a problem domain or solution-domain entity.

(ii) Action Abstraction – An object that provides a generalized set of operations, all of which perform the same kind of function.

(iii) Virtual Machine Abstraction – An object that groups together operations that are all used by some superior level of control or operations that all use some junior-level set of operations.

(iv) Coincidental Abstraction – An object that packages a set of operations that have no relation to each other.

Encapsulation – Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour. Encapsulation serves to separate the contractual interface of an abstraction and its implementation.

Encapsulation and abstraction are complementary concepts – abstraction focuses upon the observable behaviour of an object, whereas encapsulation focuses upon the implementation that gives rise to this behaviour.

Encapsulation is most often achieved through information hiding, wh is the process of hiding all the secrets of an object that do not contribute to essential characteristics, typically, the structure of an object is hidden, as w as the implementation of its methods.

It provides explicit barriers among different abstractions and thus le to clear separation of concerns.

Modularity – Modularity is the property of a system, that has be decomposed into a set of cohesive and loosely coupled modules.

The act of partitioning program into individual components can redu its complexity to some degree. Although partitioning a program is helpf more powerful justification for partitioning program is that it creates a numb of well defined, documented boundaries within the program. The boundari or interfaces, are invaluable in the comprehension of the program.

Modularization consists of dividing a program into modules which can compiled separately, but which have connections with other modules.

The connections between modules are the assumptions which the modul make about each other.

Object-orientation provides modularity, it also provides the assum benefits of modularity, namely –

(i) Reusability – Programs can be assembled from pre-writt software components that can be used in many different applications.

(ii) Extensibility – New software components can be written developed from existing ones without affecting the original components.

Meyer gives five criteria for modularity. These criteria may be applied any type of system, but are difficult to achieve using functional decomposition

(i) Decomposition – Breaking a system down into manageable uni

(ii) Composability – Modules may be freely combined into oth system.

(iii) Understandability – Understanding the part contributes fo understanding the whole.

(iv) Continuity – Small changes to the system imply small change to behaviour.

(v) Protection – Exception and error conditions are confined to th module in which they occur or affect only closely related modules.

All these criteria are met in an object-oriented system by virtue of object reflecting semantically cohesive units with simple interfaces, hiding their interna implementation. In summary, modualrity is easier to achieve using object oriented techniques, and it is therefore, easier to take full advantages of it benefits.

Hierarchy – Hierarchy is a ranking or ordering or abstraction.

Abstraction is a good thing, but in all except the most trivial applications, we may find many more different abstraction that we can comprehend at one time. Encapsulation helps manage this complexity by hiding the inside view of our abstractions. Modularity helps also, by giving us a way to cluster logically related abstractions. Modularity helps also, by giving us a way to cluster logically related abstractions. Still, this is not enough. A set of abstractions often forms a hierarchy, and by identifying these hierarchies in our design. We greatly simplify our understanding of the problem.

The two important hierarchies in a complex system are –

(i) Its class structure (the "*is a*" hierarchy)

(ii) Its object structure (the "*part of*" hierarchy).

Inheritance is the most important "is a" hierarchy, and it is an essential element of object-oriented systems. Inheritance defines a relationship among classs where in one class shares the structure or behaviour defined in one or more classes. Thus inheritance represents a hierarchy of abstractions, in which a subclass inherits from one or more subclasses. Typically, a subclass augments or redefines the existing structure and behaviour of its superclasses.

"*is a*" hierarchy denotes generalization/specialization relationship, "*part of*" hierarchy describes aggregation relationships.

Typing – Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways.

The concept of a type derives primarily from the theories of abstract data types. A type is a precise characterization of structural or behavioural properties which a collection of entities all share.

The concepts of type and class are similar, we include typing as a separate element of object model because the concept of a type places a very emphasis upon the meaning of abstraction.

Typing expresses our abstraction so that the programming language in which we implement them can be made to enforce design decisions.

Concurrency – Concurrency is the property that distinguishes an active object form one that is not active.

For certain kinds of problems, an automated system may have to handle many different events simultaneously. Other problems may involve so much computation that they exceed the capacity of any single processor. In each of these cases, it is natural to consider distributed set of computers for the target implementation or to use processors capable of multitasking.

A single process is the root from which independent dynamic action occurs within a system. Every program has at least one thread of control, but a system which involves concurrency may have many such threads, some are transitory, and others that last the entire lifetime of the system's execution.

Systems executing across multiple CPUs allow for truly concurrent th of control, whereas systems running on a single CPU can only achieve illusion of concurrent threads of control, usually by means of some t slicing algorithm.

We distinguish between heavyweight and lightweight concurrenc heavy-weight process is one that independently managed by the target opera system, and so encompasses its own address space. A lightweight proc usually lives within a single operating system process along with ot lightweight processes, which share the same address space. Communica among heavyweight processes is generally expensive, involving some form inter-process communication, and the communication among lightwe processes is less expensive, and often involves shared data.

Persistence – Persistence is the property of an object through which existence transcends time (i.e. the objects continues to exist after its crea ceases to exist) and/or space (i.e. the object continues to exist after its crea ceases to exist) and/or space (i.e. the object's location moves from the addr space in which it was created).

An object in software takes up some amount of space and exists fo particular amount of time. The spectrum of object persistence encompas the following –

(i) Transient results in expression evaluation.

(ii) Local variables in procedure activations.

(iii) Own variables, global variables, and heap items whose exten different from their scope.

(iv) Data that exists between executions of a program.

(v) Data that exists between various versions of a program.

(vi) Data that outlines the program.

The concept of persistence to the object model gives rise to obje oriented databases. Very few object-oriented programming languages provi direct support for persistence. Persistence is achieved through a mode number of commercially available object-oriented database. Anoth reasonable approach to persistence is to provide an object-oriented skin ov a relational database.

Persistence deals with more than just the life time of data. In obje oriented databases, not only does the state of an object persist, but its cla must also transcend any individual program, so that every program interpr this saved state in the same way. This clearly makes it challenging to mainta the integrity of a database as it grows, particularly if we must change the cla of an object.

Q.20. Discuss the formatted I/O operations in detail.

Or

What are the two ways to formatting output information in C++ ?

Ans. C++ supports many features that could be used for formatting the output. These features include –

(i) ios class functions and flags

(ii) Manipulators

(iii) User-defined output functions.

ios Class Functions – The ios class contains a large number of member functions that would help us to format the output in many ways. The most important ones among them are shown in table 1.1.

Table 1.1

Function	*Task*
width( )	To specify the required field size for displaying an output value.
precision( )	To specify the number of digits to be displayed after the decimal point of a float value.
fill( )	To specify a character that is used to fill the unused portion of a field.
setf( )	To specify format flags that can control the form of output display (such as left-justification and right-justification).
unsetf( )	To clear the flags specified.

Manipulators – Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream.

To access the manipulators, the file ***iomanip*** should be included in the program. Some important manipulator functions are shown in table 1.2 –

Table 1.2

Manipulators	*Equivalent ios Function*
setw( )	width( )
setprecision( )	precision( )
setfill( )	fill( )
setiosflags( )	setf( )
resetiosflags( )	unsetf( )

Width( ) – It is used for specifying the field size to display an output. The syntax is –

```
cout.width(w);
```

where w is the field width (number of columns). The output will be printed in a field of w characters wide at the right end of the field.

For example,

```
int   a = 4000, b = 600, c = 3000;
cout.width(10);
cout << a << endl;
cout.width(10);
cout << b<< endl;
cout.width(10);
cout << c<< endl;
```

Note that the field should be specified for each item separately. C++ never truncates the values and, therefore, if the specified field width is smaller than the size of the value to be printed, C++ expands the field to fit the value.

For example,

```
int a = 4852;
cout.width(2);
cout << a;
```

Output will be –

4852

Precision( ) – It is used to specify the number of digits to be displayed after the decimal point of the float value. The syntax is –

```
cout.precision(d);
```

where d is the number of digits to the right of the decimal point.

For example,

```
cout.precision(3);
cout << sqrt (2) << "\n";
cout << 3.14159 << "\n";
cout << 2.50032 << "\n";
```

will produce the following output –

1.141 (truncated)
3.142 (rounded to the nearest cent)
2.5 (no trailing zeros)

We can also combine the field specification with the precision setting.

For example,

```
cout.precision(2);
cout.width(5);
cout << 1.2345;
```

The output will be

	1		2	3

Fill( ) – It is used to specify a character that is used to fill the unused portion of a field. The syntax is –

```
cout.fill(ch);
```

where ch represents the character which is used to fill the unused portion.

For example,

```
cout.fill('*');
cout.width(10);
cout << 5250 << "\n";
```

The output would be

*	*	*	*	*	*	5	2	5	0

Setf( ) – It is used to specify the format flags that can control the form of output displayed (such as left-justification and right-justification). The syntax is –

```
cout.setf(arg1, arg2)
```

where arg1 is one of the formatting flags defined in the class ios and arg2 is a bit field specifies the group to which the formatting flag belongs. Formatting flag specifies the format action required for the output.

For example,

```
cout.setf(ios : : left, ios : : adjustfield);
cout.setf (ios: :scientific, ios: :floatfield);
```

The table 1.3 shows the flags and bit fields for setf( ) function.

Table 1.3

Format Required	*Flag (arg1)*	*Bit-field (arg2)*
Left-justified output	ios :: left	ios :: adjustfield
Right-justified output	ios :: right	ios :: adjustfield
Padding after sign or base indicator (like + ##20)	ios :: internal	ios :: adjustfield
Scientific notation	ios :: scientific	ios :: floatfield
Fixed point notation	ios :: fixed	ios :: floatfield
Decimal base	ios :: doc	ios :: basefield
Octal base	ios :: oct	ios :: basefield
Hexadecimal base	ios :: hex	ios :: basefield

Managing Output with Manipulators – The header file iomanip provides a set of functions called manipulators which can be used to manipulate the output formats. For example, two or more manipulators can be used in one statement as follows –

```
cout<< manip1 << manip2 << manip3 << item;
cout<< manip1 << item1 << manip2 << item2;
```

MsInventer

We can use manipulators with the ios functions as follows –

```
cout.setf (ios :: showpoint);
cout.setf (ios :: showpos);
cout << setprecision (4);
cout << setiosflags (ios :: scientific);
cout << setw (10) << 123.45678
```

Custom/user-defined Manipulators – We can design our o (customized) manipulators for certain special purposes. The syntax for creating a custom manipulator is shown in fig. 1.7.

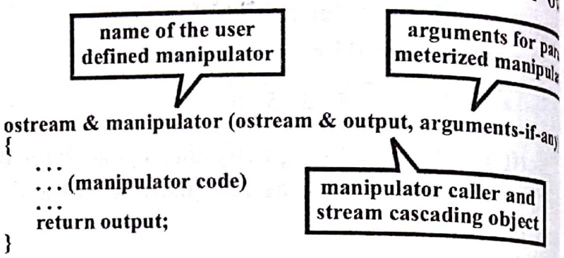


Fig. 1.7 Syntax of Creating a Custom Manipula

The following function defines a manipulator called *unit* that display "inches" –

```
ostream & unit (ostream & output)
{
    output <<"inches";
    return output;
}
```

The statement

```
cout << 36 << unit;
```

will produce the following output

36 inches

Q.21. What are C++ streams ? Explain with the stream classes.

Or

Write short note on I/O stream. ***(R.G.P.V., June 2002, Dec. 20***

Or

Define a stream. What are the three streams used for inputting, outputti and both for inputting and outputting ? ***(R.G.P.V., June 2010, 20***

Or

Explain the concept of streams in C++ and give the hierarchy of differe types of stream. ***(R.G.P.V., June 20***

Or

What are input and output streams ? Explain them with illustratio ***(R.G.P.V., June 20***

Or

What are input and output streams ? Give example. ***(R.G.P.V., June 20***

Or

What are input and output streams ? ***(R.G.P.V., June 20***

Or

Write and explain I/O stream hierarchy in C++. ***(R.G.P.V., Dec. 20***

Ans. A stream is a sequence of bytes. It acts as a ***source*** from which the input data can be obtained or as a ***destination*** to which the output data can be sent. The source stream that provides data to the program is called the ***input stream*** and the destination stream that receives output from the program is called the ***output stream***.

In other words, a program extracts the bytes from an input stream and inserts bytes into an output stream as shown in fig. 1.8.

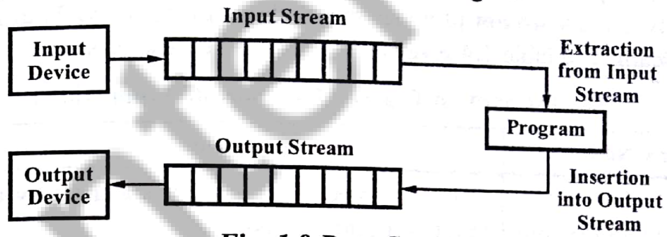


Fig. 1.8 Data Stream

The data in the input stream can come from the keyboard or any other storage device. A stream acts as an interface between the program and the I/O device. Therefore, a C++ program handles data (I/O) independent of the device used.

C++ contains several predefined streams that are automatically opened when a program begins its execution. These include cin and cout, where cin represents the input stream connected to the standard input device (usually the keyboard) and cout represents the output stream connected to the standard output device (usually the screen).

Stream Classes – The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes.

The hierarchy of the stream classes used for input and output operations with the console unit. These classes are declared in the header file *iostream*. This file should be included in all the programs that communicate with the console unit.

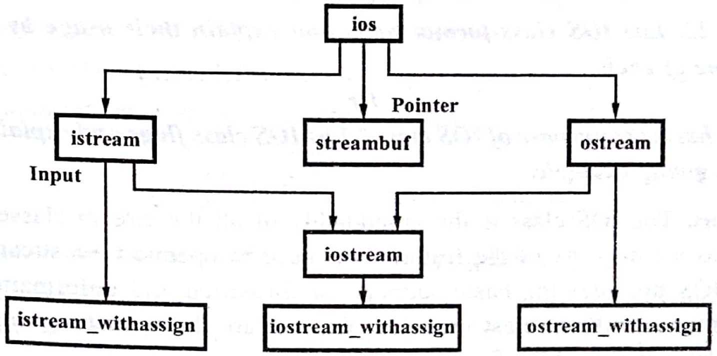


Fig. 1.9 Stream Classes for Console I/O Operations

In fig. 1.9, *ios* is the base class for ***istream*** (input stream) and *ostre* (output stream) which are base classes for ***iostream*** (input/output strea The class *ios* is declared as the virtual base class so that only one copy of members are inherited by the ***iostream***.

The *ios* class provides the basic support for formatted and unforma I/O operations. The *istream* class provides the facilities for formatted a unformatted input while the ***ostream*** class provides the facilities for forma output. The class ***iostream*** provides the facilities for handling both input a output stream. The table 1.4 gives the details about these classes.

Table 1.4 Stream Classes for Console Operations

Class Name	*Contents*
ios (General input/output stream class)	Contains basic facilities that are used by all oth input and output classes. Also contains a pointer to a buffer object (*streambuf* objec Declares constants and functions that are necessa for handling formatted input and output operation
istream (input stream)	Inherits the properties of ios Declares input functions such as get( ), getline(and read( ) Contains overloaded extraction operator >>
ostream (output stream)	Inherits the properties of ios Declares output functions *put*( ) and *write*( ) Contains overloaded insertion operator <<
iostream (input/output stream)	Inherits the properties of *ios istream* and *ostrea* through multiple inheritance and thus contains a the input and output functions
streambuf	Provides an interface to physical devices through buffe Acts as a base for filebuf class used ios files.

Q.22. List IOS class format flags and explain their usage by givi example of each.

Or

What is the purpose of IOS class ? List IOS class flags and explain the use by giving example.

Ans. The IOS class is the granddaddy of all the stream classes, an contains the majority of the features you need to operate C++ stream. Th class IOS provides the basic support for formatted and unformatted I/ operations. The three most important features are the formatting flags, th error-status flags and the file operation mode.

Formatting flags are a set of enum definitions in IOS. They act as on/off switches that specify choices for various aspects of input and output format and operation. The formatting flags are shown in table 1.5.

Table 1.5

Flag	*Meaning*
skipws	Skip (ignore) whitespace on input
left	Left adjust output [12.34]
right	Right adjust output [12.34]
internal	Use padding between sign or base indicator and number [+ 12.34]
dec	Convert to decimal
oct	Convert to octal
hex	Convert to hexadecimal
boolalpha	Convert bool to "true" "or" "false" strings
showbase	Use base indicator on output (O for octal, Ox for hex)
showpoint	Show decimal point on output
uppercase	Use uppercase X, E, and hex output letters (ABCDEF) – the default is lowercase
showpos	Display + before positive integers
scientific	Use exponential format on floating-point output [9.1234E2]
fixed	Use fixed format on floating-point output [912.34]
unitbuf	Flush all streams after insertion
stdio	Flush stdout, stderror after insertion

There are several ways to set the formatting flags, and different ones can be set in different ways. All the flags can be set using the setf( ) and unsetf( ) IOS member functions. Look at the following example –

```
cout.setf (ios :: left);   // left justify output text
cout >> "This text is left-justified";
cout.unsetf (ios :: left);   // return to default
                             // right justified
```

Many formatting flags can be set using manipulators.

Q.23. How can we create our own manipulators ?

Ans. We can create our own manipulators for certain special purposes. The general form for creating a manipulator without any arguments is as follows –

```
ostream & manipulator(ostream & output)
{
     .........
     ......... (code)
     .........
     return output;
}
```

Here, the manipulator is the name of the manipulator under creation, example –

```
ostream & currency(ostream & output)
{
     output << "Rs";
     return output;
}
```

If we use the statement

cout << 100 << currency;

then the output will be

100 Rs

Q.24. Give name and description of various fstream classes. G examples of its usage.

Ans. The I/O system of C++ has a set of classes that define the f handling methods. These include ifstream, ofstream and fstream. These clas are derived from corresponding iostream class. The fig. 1.10 shows stre class for file operations.

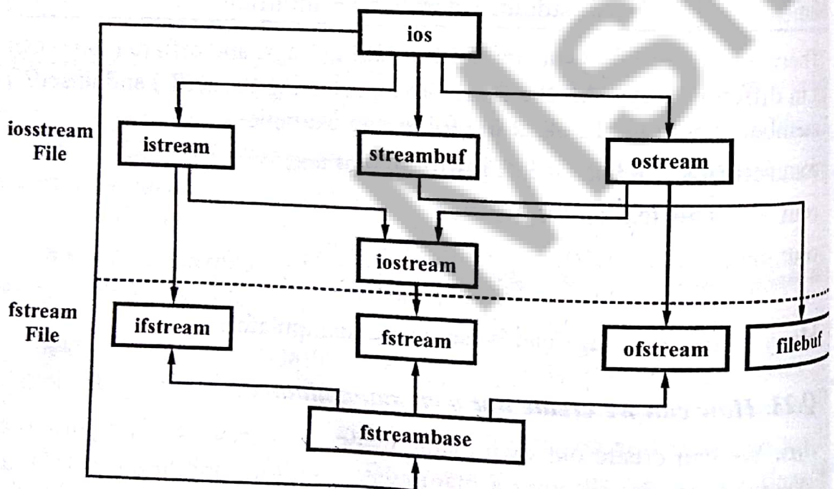


Fig. 1.10 Stream Classes for File Operations (Contained in fstream Fil

These classes, designed to manage the disk files, are declared in fstream and therefore we must include this file in any program that uses files.

The table 1.6 shows the details of file stream classes.

Table 1.6

Class	*Contents*
filebuf	Its purpose is to set the file buffers to read and write. Contains Openprot constant used in the open( ) of file stream classes. Also contains close( ) and open( ) as members.
fstreambase	Provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream classes. Contains open( ) and close( ) functions.
ifstream	Provides input operations. Contains open( ) with default input mode. Inherits the functions get( ), getline( ), read( ), seekg( ) and tellg( ) functions from istream.
ofstream	Provides output operations. Contains open( ) with default output mode. Inherits put( ), seekp( ), tellp( ), and write( ), functions from ostream.
fstream	Provides support for simultaneous input and output operations. Contains open( ) with default input mode. Inherits all the functions from istream and ostream classes through iostream.

Example –

```
#include <fstream.h>
#include <cstdlib.h>
void error(const char*p, const char*p2 = " ")
{
     cerr << p << ' ' << p2 << '\n';
     std : : exit(1);
}
int main(int argc, char*argv[ ])
{
     if (argc ! = 3) error ("wrong number of arguments");
     std : : ifstream from (argv[1]);          // open input file
                                               // stream
     if (!from) error ("cannot open input file", argv[1]);
     std : : ofstream to (argv[2]);            // open output
                                               // file stream
     if (!to) error ("cannot open output file", argv[2]);
     char ch;
     while (from.get(ch) to.put (ch);          //copy
                                               //characters
     if (!from.eof( ) // !to) error ("something strange happened");
}
```

Q.25. Write in brief about stream and types of streams. Give name, description of various fstream methods. ***(R.G.P.V., Dec. 2011, June 20***

Ans. **Stream and Types of Stream** – Refer to Q.21.

fstream Methods – Refer to Q.24.

Q.26. Write short note on ostream.

Ans. An ostream (output stream) is a mechanism for converting val of various types into sequence of characters. Usually, these characters then output using lower-level output operations. The ostream class hand output or insertion activities. The following table shows the most commo used member functions of this class. The last four functions in table 1.7 specifically with disk files.

Table 1.7

Function	***Purpose***
<<	Formatted insertion for all basic (and overloaded) types.
put(ch)	Insert character ch into stream.
flush( )	Flush buffer contents and insert newline
write(str, SIZE)	Insert SIZE characters from array st into file.
seekp(position)	Set distance in bytes of file pointer from start of file.
seekp(position, seek_dir)	Set distance in bytes of file pointer, from specified place in file. seek_dir can be ios : : beg, ios : : cur, or ios : : end.
pos = tellp( )	Return position of file pointer, in bytes.

Q.27. Write short note on extractors and inserters.

Ans. It is possible to overload the I/O operators relative to a cl that we create. When we overload << operator for output, we creating an inserter function or simply *inserter.* An inserter has following form –

```
ostream & operator << (ostream & stream, class-name object);
{
    .........
    ......... // Body of inserter return stream;
}
```

An *inserter* returns a reference to stream, which is of type *ostream.*

Similar to inserters, we can create *extractors* by overloading the input >> operator relative to a class in which it is used. The general form of an extractor function is as follows –

```
istream & operator >> (istream & stream, class-name & object);
{
    .........
    ......... // Body of extractor return stream;
}
```

Extractors return a reference to stream, which is of type istream.

Q.28. Write short note on files.

Ans. A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files.

The I/O system of C++ handles file operations which are very much similar to the console input and output operations. It uses file streams as an interface between the programs and the files. The stream that supplies data to the program is known as input stream and the one that receives data from the program is known as output stream.

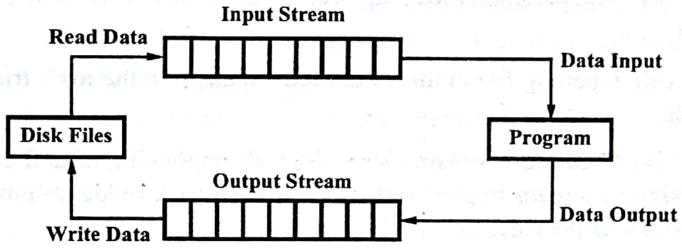


Fig. 1.11 File Input and Output Streams

The input operation involves the creation of an input stream and linking it with the program and the input file. Similarly, the output operation involves establishing an output stream with the necessary links with the program and the output file.

Q.29. How explain the stream and files in object oriented programming ?

Ans. Refer to Q.21 and Q.28.

Q.30. What is file mode ? Describe the various file mode options available. ***(R.G.P.V., Dec. 2016)***

Ans. File mode specifies the purpose for which the file is opened. The general form of opening file with the file mode is as follows –

stream-object.open ("filename", mode);

MSinverter

The file mode parameter can take one or more constants defined in class *ios*. The table 1.8 shows the file mode parameters.

Table 1.8

Parameter	*Meaning*
ios :: app	Append to end-of-file
ios :: ate	Go to end-of-file on opening
ios :: binary	Binary file
ios :: in	Open file for reading only
ios :: nocreate	Open fails if the file does not exist
ios :: noreplace	Open files if the file already exists
ios :: out	Open file for writing only
ios :: trunc	Delete the contents of the file if it exi

(i) ios :: app and ios :: ate take us to the end of the file when opened. The difference between the two parameters is that the ios :: allows us to add data to the end of the file only, while ios :: ate mode per us to add data or to modify the existing data anywhere in the file.

(ii) The parameter ios :: app can be used only with the files cap of output.

(iii) Opening file in ios :: out mode opens it in the ios :: trunc m by default.

(iv) Creating a stream using ifstream implies input and creati stream using of stream implies output. In these cases, mode parameters not necessary to provide.

(v) The fstream class does not provide a mode by default, so must provide the mode explicitly when using an object of fstream class.

(vi) The mode can combine two or more parameter using bitwis operator (symbol |) as follows –

fout.open ("data", ios :: app | ios :: nocreate)

Q.31. Explain the process of opening and closing a file.

Or

Describe how can files be opened and closed explicitly in a progr

(R.G.P.V., June 2011, 20

Or

What is meant by initializing a file stream object ? What are the w of doing it ? Give example code for each of them. (R.G.P.V., June 20

Ans. A file can be opened in two ways –

(i) Using the constructor function of the class.

(ii) Using the member function open( ) of the class.

The first method is useful when we use only one file in the stream. The second method is used when we want to manage multiple files using one stream.

A constructor is used to initialize an object during its creation. Here, a filename is used to initialize the file stream object. The creation and assignment of file name to the file stream object involves the following steps –

(i) Create a file stream object using the appropriate class depending on the type of the file stream required.

(ii) Initialize the file object with the desired filename.

With the first method, we open the file for output

ofstream outfile("result"); //output only

and similarly, for input

ifstream infile("data"); //input only

In the above statements, the outfile is created as an ofstream object that manages the output stream. This opens the "result" file and attached it to the output stream outfile. The infile is declared as an ifstream object and attached it to the file data for reading (input).

With the second method, files can be opened using function open( ) to open multiple files that use the same stream object.

For example, if we want to process a set of files sequentially, we may create a single stream object and use it to open each file as follows –

```
file-stream-class stream-object;
stream-object.open("filename");
```

The connection with a file is closed automatically when the stream object expires (when the program terminates).

We can use to disconnect the files the statements as follows –

```
............
............
    outfile.close( ); //disconnect from outfile
    ifstream infile("salary");
............
............
    infile.close( );  //disconnect from infile
```

MSinverter

Q.32. *Explain the file pointers and the function for the manipula of the file pointers.*

Ans. Each file has two associated pointers that is called the *file poin* One of them is input pointer or get pointer and the other is output point put pointer. We can use these pointers to move through the files while rea or writing.

When we open a file in read-only mode, input pointer automatically the beginning so that we can read the file from start. When we open a fi write-only mode, the existing contents are deleted and the output pointer to the beginning. This enables us to write to the file from the start.

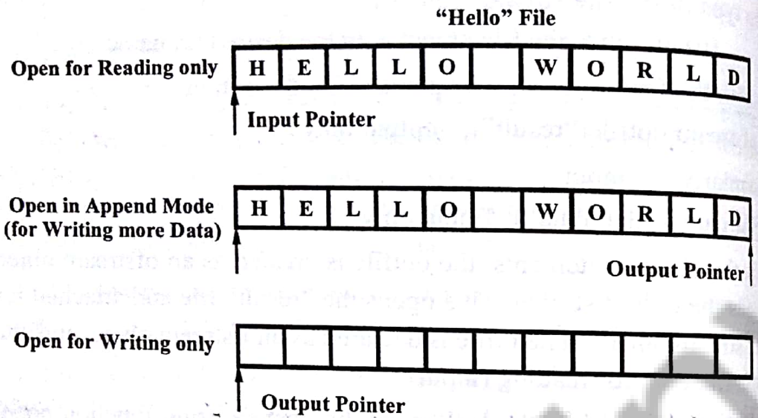


Fig. 1.12 Action for File Pointer While Opening a File

If we want to open an existing file to add more data, the file is ope 'append' mode. This moves the output pointer to the end of the file (i.e., of the existing contents).

Functions for Manipulation of File Pointers – All the actions on pointers take place automatically by default. The file stream classes sup the following functions to manipulate the situations –

(i) ***seekg( )*** – It moves get pointer (input) to a specified locati
(ii) ***seekp( )*** – It moves put pointer (output) to specified locati
(iii) ***tellg( )*** – It gives the current position of the get pointer.
(iv) ***tellp( )*** – It gives the current position of the put pointer.

For example,

infile.seekg(10);

It moves the file pointer to the byte number 10. Note that, the bytes i file are numbered beginning from zero. Therefore, the pointer will be point to the 11th byte in the file.

Now, consider the following statements –

ofstream fileout;

fileout.open ("hello", ios : : app);

int p = fileout.tellp( );

On the execution of these statement, the output pointer is moved to the end of the file "hello" and the value of p will represent the number of bytes in the file.

Seek functions *seekg*( ) and *seekp*( ) can also be used with two arguments as follows –

seekg(offset, refposition);

seekp(offset, refposition);

The parameter *offset* represents the number of bytes the file pointer is to be moved from the location specified by the parameter *refposition*.

The *refposition* takes one of the following three contents defined in the *ios* class –

ios : : beg	Start of the file
ios : : cur	Current position of the pointer
ios : : end	End of the file

The *seekg*( ) function moves the associated file's 'get' pointer while *seekp*( ) function moves the associated file's put pointer.

The table 1.9 lists some sample pointer offset calls and their actions. *fout* is an *ofstream* object.

Table 1.9

Seek Call	*Action*
fout.seekg(o, ios :: beg);	Go to start
fout.seekg(o, ios :: cur);	Stay of the current position
fout.seekg(o, ios :: end);	Go to the end of file
fout.seekg(m, ios :: beg);	Move to (m + 1)th byte in the file
fout.seekg(m, ios :: cur);	Go forward by m byte from the current position
fout.seekg(-m, ios :: cur);	Go backward by m bytes from the current position
fout.seekg(-m, ios :: end);	Go backward by m bytes from the end.

Q.33. What are the functions supported by file stream classes performing I/O operations ? ***(R.G.P.V., June 2009, Dec. 2***

Or

Illustrate the functions of the ifstream and ofstream class for chara input and output and numeric input and output with examples. ***(R.G.P.V., Dec. 20***

Ans. The file stream classes support a number of member functions performing the input and output operations on files. put( ) and get( ) functi are designed for handling a single character at a time. write( ) and rea functions are designed to write and read blocks of binary data.

put( ) and get( ) Functions – The function put( ) writes a single chara to the associated stream. Similarly, the function get( ) reads a single chara from the associated stream. The following program shows how these functi work on a file. First, the program requests for a string. On receiving string, the program writes it, character by character, to the file using the p function in a **for** loop. **for** loop is terminated by using the length of the st Then, the program displays the contents of the file on the screen. The g function is used to fetch a character from the file and continues to do so u the end-of-file condition is reached. The character read from the file is displ on the screen using the operator <<.

```
//I/O Operations on Characters
#include <iostream.h>
#include <fstream.h>
#include <string.h>
int main( )
{
    char string[80];
    cout <<"Enter a string \n";
    cin >> string;
    int len = strlen(string);
    fstream file;                  // input and output stream
    file.open("TEXT", ios :: in | ios :: out);
    for(int i=0; i<len; i++)
    file.put(string[i]);           // put a character to file
    file.seekg(0);                 // go to the start
    char ch;
    while(file)
    {
        file.get(ch);              // get a character from file
        cout << ch;                // display it on screen
    }
    return 0;
}
```


write( ) and read( ) Functions – The functions write( ) and read( ), unlike the functions put( ) and get( ), handle the data in binary form. This means that the values are stored in the disk file in the same format in which they are stored in the internal memory. Fig. 1.13 shows how an int value 2594 is stored in the binary and character formats. An *int* takes two bytes to store its value in the binary form irrespective to its size. But a 4 digit int will take four bytes to store it in the character form. The binary format is more accurate for storing the numbers as they are stored it in the character form.

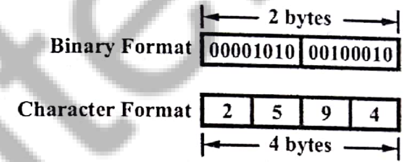


Fig. 1.13 Binary and Character Formats of an Integer Value

The binary format is more accurate for storing the numbers as they are stored in the exact internal representation. There are no conversions while saving the data and therefore saving is much faster. The binary input and output functions takes the following form –

infile.**read**((char*) &V, sizeof (V));

outfile.**write**((char*) &V, sizeof (V));

These functions take two arguments. The first is the address of the variable V, and the second is the length of that variable in bytes. The address of the variable must be cast to type char* (i.e., pointer to character type). Program given below shows how these two functions are used to save an array of float numbers and then recover them for display on the screen.

```
//I/O Operations on Binary Files
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
const char * filename = "BINARY";
int main( )
{
    float height[4] = {175.5, 153.0, 167.25, 160.70};
    ofstream outfile;
    outfile.open(filename);
    outfile.write((char*) & height, sizeof(height));
    outfile.close( );           // close the file for reading
```

Asinventer

```
    for(int i=0; i<4; i++)          // clear array from memory
        height[i] = 0;
    ifstream infile;
    infile.open(filename);
    infile.read((char *) & height, sizeof(height));
    for(i=0; i<4; i++)
    {
        cout.setf(ios::showpoint);
        cout << setw(10) << setprecision(2)
             << height[i];
    }
    infile.close( );
    return 0;
}
```

The output of Program would be –

175.50   153.00   167.25

UNIT 2

ENCAPSULATION AND DATA ABSTRACTION, CONCEPT OF OBJECTS – STATE, BEHAVIOUR AND IDENTITY OF AN OBJECT

Q.1. Write short note on encapsulation in OOPs. (R.G.P.V., June 2015)

Ans. The process of compartmentalizing the elements of an abstraction that constitute its structure and behaviour is known as encapsulation; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

Encapsulation is the wrapping up of data and function into a single unit. The most important feature of class is data encapsulation; because data is not accessible to the outside world and only those functions which are wrapped in the class can access it. This insulation of data from direct access by the program is referred to as data hiding or information hiding.

A process of including in an object everything it needs, hidden from other objects in the system is encapsulation. The internal state of an object is not directly accessible from outside and cannot be changed by external changes to the application. Likewise, changes to the internal implementation details are made without affecting the external interface.

Q.2. What are the advantages of encapsulation ?

Ans. The object-oriented class and the objects drawn from class encapsulate data and the operation that work on the data in a single package. This offers us following advantages –

(i) The internal implementation details of data and procedures are hidden from outside world. This minimizes the propagation of side effects when changes occur.

(ii) Data structure and the operations that manipulate them are merged in a single named entity-class. This allows component reuse.

(iii) Interfaces among encapsulated objects are simplified. An object that sends a message does not require details of internal data structures. Hence, interfacing is simplified and the system coupling tends to be minimized.

MSinventer

Q.3. What is data hiding ? How it is done in object oriented progr-ming ? What are the advantages of data hiding ? ***(R.G.P.V., June 20***

Or

What do you mean by information hiding ? ***(R.G.P.V., June 20***

Ans. The wrapping up of data and functions into a single unit called is encapsulation. The data is not accessible to the outside world, and those functions which are wrapped in the class can access it. These funct provide the interface between the objects data and the program. This insula of the data from direct access by the program is called ***data hiding information hiding.***

The primary mechanism for hiding data is to put it in a class and ma private. Private data or functions can only be accessed from within the c whereas public data or functions are accessible from outside the class.

Data hiding means hiding the data from parts of the program that do need to access it. More specifically, one class's data is hidden from ot classes. Data hiding is designed to protect well intentioned programmers f honest mistakes.

Programmers who really want to access private data but they will fi hard to do so by accident. This is the significance of data hiding.

Also, data hiding helps to achieve encapsulation which is the proces hiding all the secrets of an object that do not contribute to its essen characteristics; typically, structure of objects and implementation of its meth is hidden.

Advantages of Data Hiding – Refer to Q.2.

Q.4. Define reusability and data hiding. ***(R.G.P.V., Dec. 20***

Ans. **Reusabiliy** – In object oriented programming, the concept inheritance provides the idea of reusability. This means that we can add additio features of an existing class without modifying it. This is possible by deriv a new class from the existing one. The new class will have the combi features of both the classes. The real appeal and power of the inherita mechanism is that it allows the programmer to reuse a class that is almost, not exactly, what we wants, and to tailor the class in such a way that it d not introduce any undesirable side effects into the rest of the classes.

Data Hiding – Refer to Q.3.

Q.5. Define and explain abstraction.

Or

Write short note on data abstraction. ***(R.G.P.V., Dec. 20***

Ans. Abstraction refers to the act of representing essential features with including the background details or explanations. Classes use the concept

abstraction and are defined as a list of abstract attributes such as size, weight and cost, and function to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called data members, because they hold information. The functions that operate on these data are sometimes called methods or member functions. Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT).

The process of deciding what parts of a real world object are essential to modeling the solution to a particular problem is called abstraction. Abstraction is the representation of all the essential features of an object, which means its possible states and behaviours. These are 'encapsulated' into an 'abstract data type' which define how all objects in class (type) of object are to be represented and how they behave.

Abstraction is a mechanism that enables the designer to focus on the essential details of a program component with little concern for lower-level detail. Abstraction is a relative concept. As we move to higher levels of abstraction we ignore more and more details, i.e., we give a more general look of a concept or item. We give a more specific view of a concept or item, as we move to lower levels of abstraction, we introduce more details.

Abstraction permits one to concentrate on a problem at some level of generalization without regard to irrelevant low-level details; use of abstraction also permit one to work with concepts and terms that are familiar in the problem environment.

At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken. Problem oriented terminology is coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

Q.6. Explain about abstract data types with proper examples. ***(R.G.P.V., Dec. 2011)***

Or

Explain abstract data type with example. What are its advantages ? ***(R.G.P.V., Dec. 2015)***

Or

What are abstract data types ? ***(R.G.P.V., June 2016)***

Or

Define abstract data types. ***(R.G.P.V., June 2014)***

Ans. The term data type defines the type of data which a particular variable can hold – it may be an integer, a character, a float, or any of a range of simple data storage representations. However, when we build object-oriented systems,

we use more complex data types, known as ***abstract data types,*** which repres more realistic entities. For example, we might be interested in representing 'bank account' data type, which describes how all bank accounts are handl in a program.

Abstraction is about reducing complexity, ignoring unnecessary deta An abstract representation of something is supposed to contain the essen features of what is being represented. For example, realistic painting conta all the details seen by the eye, whereas an abstract painting intends to refle the essential features of what is being seen.

Another key point about an abstract data type is that it represents wha common between all examples of that particular thing, not just an individu example. An abstract data type for 'coffee cup' would be a representation all the things that are common to all coffee cups, not just a particular one means that an abstract data type contains no values, it only defines the typ of data or data structures which together will define the state of any object that type.

Advantages of Abstract Data Types – The advantages of abstract da types are given below –

(i) Representation Independence – Most of the program co becomes independent of the abstract data type's representation, so th representation can be update without breaking the full program.

(ii) Modularity – With representation independence the differe parts of a program code become less dependent on other parts and on ho those other parts are implemented.

(iii) Interchangeability of Parts – Different implementations of t abstract data type can have different performance features with abstract da types, it becomes easier for each part of a program code to use an implementatio of its data types that will be more efficient for that specific part of the progra

Q.7. What type of abstraction is employed in
(i) Procedure
(ii) Object-oriented programming ?

(R.G.P.V., June 200

Ans. Table 2.1 lists the programming styles with the different types o abstraction they employ.

Table 2.1 Types of Programming Paradigms

S.No.	Programming Style	Abstraction Employed
(i)	Procedure-oriented	Algorithms
(ii)	Object-oriented	Classes and objects

Q.8. What is an object ? How can do we use objects in program ?
(R.G.P.V., Dec. 2005, Nov. 2018)

Ans. An object has property, state, behaviour, and identity. The term object and instance are interchangeable. Object is uniquely identified by its name and it defines a state which is represented by the values of its attributes at a particular time. The object's state changes on the basis of the methods which are applied to it. These possible sequences of state change are called as behaviour of the object.

The object's behaviour is defined by the set of method which can be applied on it. Property specifies current state of an object.

Generally, an object represents an individual identifiable item unit, or entity, either real or abstract, with a well defined role in problem domain. The basic run-time entities in an object oriented paradigm are called objects. They may represent a place, a person, a bank-account, a table of data or any item that is handled by the program.

Q.9. Explain different types of objects. (R.G.P.V., Dec. 2005, Nov. 2018)

Ans. There are four types of object which we may instantiate in a program. The first three are objects with specific names, but the fourth, dynamic objects, cannot be identified in this way –

(i) External (Global) Objects – An external object is one which is persistent and visible throughout a program module – i.e. its scope is an entire module (source file). It may also be made visible in other modules. Objects which fall into the category of 'external' would be one whose number and identities remain constant throughout an application.

(ii) Automatic (Local) Objects – As well as external, global objects, we may also have a number of locally declared 'automatic' objects – objects which exist in a predictable manner for a particular period of time. The key difference between an external and an automatic object is that whereas an automatic object is instantiated within the scope of part of a program module, an external object is instantiated outside of any scope (in C++ scope is defined by braces). 'Automatic' objects are automatically destroyed when they fall-out of the scope in which they were instantiated.

(iii) Static Objects – We know that external objects are persistent and visible throughout the lifetime of a program, whereas automatic objects are only persistent and visible within the scope in which they are declared. There is also the possibility in C++ to explicitly declare a variable or object which has the scope (in terms of visibility) of an automatic object but the lifetime of an external object.

(iv) Dynamic Objects – Lifetime may be controlled within a particular scope.

Q.10. How can we decide the life and scope of object in a class ?
(R.G.P.V., Dec. 201

Ans. When we create a named object in a program, its lifetime and visibi is controlled only by its scope. Named objects may be of three types –

(i) Automatic Objects – Objects instantiated inside the local sco of a function or other structure with its body defined by braces. They e exist while they are in scope.

(ii) External Objects – Objects instantiated outside any functi body. These have 'file scope' and exist for the lifetime of the program.

(iii) Static Objects – Object instantiated inside local scope and hav local visibility, but persisting from their declaration to the end of the progr

Q.11. What are global and local objects ? Also explain what do y understand by object life time. ***(R.G.P.V., June 20***

Ans. Refer to Q.9 and Q.10.

Q.12. Differentiate between global and local objects. Also write th advantages and disadvantages. ***(R.G.P.V., Dec. 20***

Ans. The difference between global and local objects are as follows-

S.No.	Global Object	Local Object
(i)	Global objects are declared outside the function bodies.	Local objects are declared inside t function bodies.
(ii)	All function can access the global objects.	Only that function in which loc object is defined can access loc objects.
(iii)	Global object does not get first precedence.	Local object gets first precedenc
(iv)	For example – #include <iostream.h> #include <constream.h> class text { public : void show (char*c) { cout << "\n" << c; } }; text g; // global object declaration void main( ) { : : g.show ("Global"); // call using global object }	For example – #include <iostream.h> #include <constream.h> class text { public : void show (char*c) { cout << "\n" << c; } }; void main( ) { text local; //local object declarati local.show ("Local"); //call using local objec }

Advantages of Global Object – There are several advantages of global objects are as follows –

(i) It can be access in whole class.

(ii) Only one global object can be used in all function declared in that class.

Disadvantages of Global Object – There are several disadvantages of global objects are as follows –

(i) If variable has declared as global it cannot declared as local with the same name in the functions.

(ii) It is time consuming means when one function does not need to the global variable yet compiler reads the global variable when object call the class.

Advantages of Local Object – There are several advantages of local objects are as follows –

(i) It can be access in function bodies in which it is declared.

(ii) Different-different local objects can be defined in different-different functions.

(iii) Local objects can be defined when function needs.

Disadvantages of Local Object – There are several disadvantages of local variable are as follows –

(i) If object has declared as local it cannot declared as global with the same name in the class and cannot declared as local in other function.

(ii) It has limited scope.

(iii) Local object of one function cannot be used in other function.

(iv) It consumes more memory because specific task has specific local variable.

Q.13. Differentiate between global and local objects.
(R.G.P.V., June 2012)

Or

Compare the global and local objects. ***(R.G.P.V., Dec. 2011)***

Ans. Refer to Q.12.

Q.14. Differentiate between static and dynamic objects.
(R.G.P.V., June 2012, 2016)

Or

Compare the static and dynamic objects.
(R.G.P.V., Dec. 2011, June 2014)

Ans. There are several differences between static and dynamic objects are given below –

S.No.	Static Object	Dynamic Object
(i)	A static object gets its memory preallocated before runtime.	A dynamic object gets it me allocated at runtime.
(ii)	If you define an array of length five as static object, its length cannot be change at runtime.	If you define an array of vari length as dynamic object length can be change at runti
(iii)	It is faster for particular set of elements.	It is not faster for particular se elements.

Q.15. What is meant by life time of an object ? Compare static dynamic objects. ***(R.G.P.V., Dec. 2***

Ans. Refer to Q.10 and Q.14.

Q.16. Explain the significance of data hiding with an example. Comp static and dynamic objects. ***(R.G.P.V., June 2***

Ans. Refer to Q.3 and Q.14.

Q.17. What is the difference between active and passive state of obje ***(R.G.P.V., Dec. 2***

Ans. Following are the differences between active and passive obje

(i) An ***active object*** is one that encompasses its own thread of con whereas a ***passive object*** does not.

(ii) Active objects are generally autonomous meaning that they exhibit some behaviour without being operated upon by another object. On other hand, passive objects can only undergo a state change when explic acted upon.

(iii) Active objects serve as the roots of control. If our system invol multiple threads of control, then we will usually, have multiple active objects the other hand, sequential systems, usually, have exactly one active object, s as a main window object responsible for managing an event loop that dispatc messages. In such architecture all other objects are passive, and their behavio ultimately triggered by messages from one active object. That is, passive ob never operates on other objects but it is only operated upon by other objects.

Q.18. What are objects ? How are they created ? (R.G.P.V., Dec. 20

Ans. **Object** – Refer to Q.8.

Once a class has been created, we can create variables of that type using the class name. For example,

```
class_name X; //memory for X is created
```

creates a variable of type class_name. In C++, the class variables are known objects. Therefore, X is called an object of type class_name. We may also decl more than one object in one statement. The declaration of an object is simila

that of a variable of any basic type. The necessary memory space is allocated to an object at this stage. Note that the class specification, like a structure, provides only a *template* and does not create any memory space for the objects.

Objects can also be created when a class is defined by placing their names immediately after the closing brace, as we do in the case of structures. That is to say, the definition

```
class   class_name
{
...............
...............
...............
...............
}
x, y, z;
```

would create the object x, y and z of type item. This practice is seldom followed because we would like to declare the object close to the place where they are used and not at the time of class definition.

Q.19. A method implement the behaviour of an object. Justify.

Ans. Method is defined for the class and objects use the method to perform an action. Each method is synonymous with a behaviour or a series of methods, which, when executed, constitutes a behaviour. The object on which method operates is called the receiver objects, operating through its attributes and methods.

Q.20. Write short note on object as software modules.

Ans. Objects are software representations of real-world entities. To capture the features and capabilities of real world, object consists of both attributes and operational information. An object is a software bundle of related variable and methods. Software objects are often used to model real-world object that we find in everyday life.

Objects are central idea behind object-oriented programming. Object-oriented programming treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it, and protect it from accidental modification from outside functions. OOP allows decomposition of problem into a number of entities called object and then builds data and functions around these objects.

In object-oriented programming (OOP), an object is a self-containing entity that consists of data and procedure to manipulate the data. Objects are the thing, we think about first in designing a program and they are also the units of code that are eventually derived from the process. An object is what actually runs on the computers.

Q.21. What is dynamic initialization of objects ? Why is it needed? How is it accomplished in C++ ? Illustrate.

Or

What is dynamic initialization of objects ? Justify its importance with example. How it is implemented in programming ? **(R.G.P.V., Dec. 20**

Or

How is dynamic initialization of objects achieved ? **(R.G.P.V., Dec. 20**

Ans. Dynamic initialization of objects refers that the initial value of object may be provided during run time. One advantage of dynamic initializa is that we can provide various initialization formats, using overloa constructors. This provides the flexibility of using different format of dat run time depending upon the situation.

Consider, the example COUNTER which provides a counter variable t can be modified only through its member functions.

```
// counter.cpp
// object represents a counter variable
#include <iostream.h>
using namespace std;
class Counter
{
private:
      unsigned int count;                   // count
public:
      Counter( ) : count(0)                 // constructor
            { /*empty body*/}
      void inc_count( )                     // increment count
            {count++;}
      int get_count( )                      // return count
            {return count;}
};
int main( )
{
Counter c1,c2;                              // define and initialize
cout << "\nc1=" << c1.get_count( );         // display
cout << "\nc2=" << c2.get_count( );
c1.inc_count( );                            // increment c1
c2.inc_count( );                            // increment c2
c2.inc_count( );                            // increment c2
cout << "\nc1=" << c1.get_count( );         // display again
cout << "\nc2=" << c2.get_count( );
return 0;
}
```

When an object of type ***Counter*** is first created, we want its count to initialized to 0. In the counter class the constructor counter does this. Th

function is called automatically whenever a new object type counter is created. Thus in main( ), the statement

Counter c1, c2;

not only creates two objects, but also initializes their count variables to 0.

Q.22. Write a program in C++ to demonstrate the creation and use of dynamic objects.

Ans. The following program demonstrates the creation and use of dynamic objects –

```
#include <iostream.h>
#include <string.h>
class car
{
      private:
            char colour[10];
      public:
            car(char* colour_in);
            char* getcolour( );
};
car : : car(char* colour_in)
{
      strncpy(colour, colour_in, 9);
      colour[9] = '\0';
}
char* car : : getcolour( )
{
      return colour;
}
int main( )
{
      car * cars[10]; // declares an array of ten 'car' object pointers
      cars[0] = new car("Red");
      cars[1] = new car("Green");
      cars[2] = new car("Yellow");
      cars[3] = new car("Blue");
      cout << cars[0] –> getcolour( );
      cout << cars[1] –> getcolour( );
      cout << cars[2] –> getcolour( );
      cout << cars[3] –> getcolour( );
      delete cars[0];
      cars[0] = NULL;
      delete cars[1];
      cars[1] = NULL;
      delete cars[2];
      cars[2] = NULL;
      delete cars[3];
      cars[3] = NULL;
}
```

MSinventer

Q.23. Write short note on object identity. **(R.G.P.V., Dec. 20**

Ans. A class is defined by three elements – a unique name, attributes a methods. In contrast, an object is defined by identity, state and behaviour.

In each of these three cases, the property of the class relates in some w to the property of the object. The attributes of the class allow each object contain state data – one value for each attribute. The methods of the cla define the possible behaviours of the object. However, the concept of ident is slightly more complex than the name of the class.

It is possible for a specific object to be identified by a unique name. F instance in an oil field control system, there might be a fixed and limit number of individual oil pumps, each one easily identifiable by a uniq name such as 'oil pump1', 'oil pump2' and so on. However, this simplici of identity is not always the case. In some situations, there are many obje constantly being created and destroyed. If it is raining, what identifies ea raindrop ? It would be absurd to suggest that we could name every raindro but each does exist and therefore must have some kind of unique identity. some cases, the idea that each object can have a unique name falls down. fact, what differentiates one raindrop from another is simply the space th it occupies at a particular time. The identity of any object may be ultimate definable only by this space time relationship. In object-oriented progra which have to keep track of a large number of objects which are constant being instantiated and destroyed, objects will be identified by their memo locations rather than by unique names, but objects which are unpredictab are identified by location.

Q.24. Write short note on persistent objects.

(R.G.P.V., Dec. 2011, 2012, June 201

Or

What do you mean by object persistent ? **(R.G.P.V., Dec. 201**

Ans. The lifetime of an object can vary from a momentary existence insi the body of a function or method, to persistence for the life of the program.

There are three levels at which objects may persist –

(i) Object persisting during a run of a program

(ii) Objects persisting between runs of a single program

(iii) Objects persisting between different programs.

Objects which only exist while a program is running are known as ***transie objects.*** They have no existence independent of a single program run tim Those whose lifetimes extend beyond the boundaries of a single program ru are known as ***persistent objects.*** This means that a persistent object is o whose data can be saved to disk.

Q.25. What is object identifier ? **(R.G.P.V., Dec. 2014)**

Ans. Object oriented languages have built-in mechanisms for identifying objects, and ways to test object identity. There is typically no need to create explicit object identifier types. If you need a unique object identifier you can obtain it at run time from the system. In C++ an object's actual memory address serves as a unique identifier and can be obtained by applying the & (address of) operator to an object or object reference. Object identity can be tested by pointer (address).

Q.26. Write a brief note on object modelling. **(R.G.P.V., Dec. 2012)**

Ans. The first step in analyzing the requirements is to construct an object model. The object model shows the static data structure of the real-world system and organizes it into workable pieces. It describes real-world object classes and their relationships to each other. The object model precedes the dynamic model and functional model because static structure is usually better defined, less dependent on application details, more stable as the solution evolves, and easier for humans to understand.

Information for the object model comes from the problem statement, expert knowledge of the application domain, and general knowledge of the real world. If the designer is not a domain expert, the information must be obtained from the application expert and checked against the model repeatedly. Object model diagrams promote communication between computer professionals and application-domain experts.

Firstly, identify classes and associations as they affect the overall structure and approach to the problem. Next add attributes to further describe the basic network of classes and associations. Then combine and organize classes using inheritance. Try to specify inheritance directly without first describing low-level classes and their attributes often distort the class structure to match preconceived notions. Add operations to classes later as a by-product of constructing the dynamic and functional models. Operations modify objects and therefore cannot be fully specified until the dynamics and functionality are understood.

Hence, the following steps are performed in constructing an object model –

(i) Identify objects and classes

(ii) Prepare a data dictionary

(iii) Identify associations between objects

(iv) Identify attributes of objects and links

(v) Organize and simplify object classes using inheritance

(vi) Verify that access paths exist for likely queries

(vii) Iterate and refine the model

(viii) Group classes into modules.

Q.27. What is the need of modeling ? ***(R.G.P.V., Dec. 20***

Ans. The process of creating a model helps an analyst clarify and ref requirements and design details. The analyst learns as he or she completes then studies parts of the model. Analyst also raise questions while creating model and answer them as the modeling process continues. New pieces added; the consequences of changes are evaluated and again questioned this respect, the modeling process itself provides direct benefit to the analy The technique used to create the model is valuable in itself, even if the anal never shows a particular model to anyone else.

Q.28. What is the use of system sequence diagram ? ***(R.G.P.V., Dec. 20***

Ans. The sequence diagram is an UML diagram used to show the dynam properties of a system. A sequence diagram shows the messages that passed between two components in a system. In order to be a strict sequen diagram, these two parts have to be objects. In the early phases of syst development the components used in the diagram can be e.g. GUI, databa servers, internets or subsystems. Sequence diagrams applied in this way called system sequence diagrams. Even though they have no formal stand in the UML 2.0 definition they are widely used in the industry.

CLASSES – IDENTIFYING CLASSES AND CANDIDATES FOR CLASSES ATTRIBUTES AND SERVICES, ACCESS MODIFIERS, STATIC MEMBERS OF A CLASS, INSTANCES, MESSAGE PASSIN AND CONSTRUCTION AND DESTRUCTION OF OBJECTS

Q.29. What is a class ? How does it accomplish data hiding ?
(R.G.P.V., Dec. 20

Ans. **Class** – A ***class*** describes a group of objects with similar properti common behaviour, common relationships to other objects and comm semantics. Person, company, animal, process and window are all object class Each person has an age, IQ and may work at a job. Each process has owner, priority and list of required resources.

Data Hiding – Refer to Q.3.

Q.30. What are empty classes ? Can instance empty class be created Give reasons.

Ans. We know that the main reason for using a class is to encapsul data and code. However, it is possible to have a class that has neither data code. That is, it is possible to have empty classes. The empty class is declar as follows –

```
class xyz { };
```


In the initial stages of development of a project, some of the classes are either not fully identified, or not fully implemented. In such situations, they are implemented as empty classes during the first few implementations of the project. These empty classes are also known as ***stubs.*** The empty classes are useful in exception handling.

Q.31. What do you understand by class declaration ? Explain.

Ans. A class is a way to bind the data and its associated functions together. It allows the data to be hidden, if necessary, from external use. When defining a class, we create a new abstract data type that can be treated like any other built-in data type. The class declaration describes the type and scope of its members.

The general form of class declaration is as follows –

```
class class_name
{
    private:
        variable declarations;
        function declarations;
    public:
        variable declarations;
        function declarations;
};
```

The class declaration is similar to a structure declaration. The body of a class is enclosed within braces and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively known as class members.

They are usually grouped under two sections, namely, private and public to denote which of the members are private and which of them are public. The keywords private and public are known as visibility labels and are followed by a colon.

The class members that have been declared as private can be accessed only from within the class, and the public members can be accessed from outside the class also.

The data hiding (using private declaration) is the key feature of object-oriented programming. The use of the keyword private is optional. By default, the members of a class are private.

The variables declared inside the class are called data members and the functions are called member functions. Only the member functions can have access to the private data members and private functions.

Example –

```
class class_data
{
    private:
        int a, b;
    public:
        void putdata(int p, int q)
        {
            a = p;
            b = q;
        }
        void show_data( )
        {
            cout << "\n a =" << a << "b =" <<b;
        }
};
```

Q.32. Explain hierarchy of class with example.

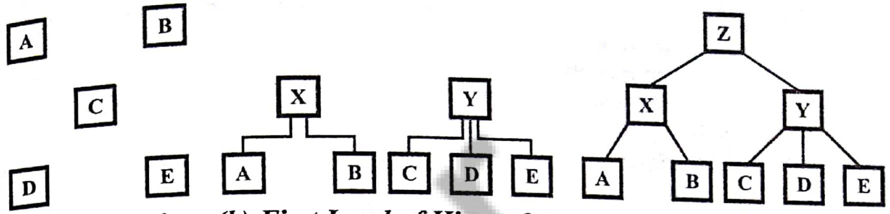
Ans. To reuse as much as data and/or functions that have been desig already, we create a class hierarchy. Class hierarchy consists of a base c and derived classes.

Organization of the class hierarchies involves identification of com attributes and functions among a group of related classes and then combi them to form a new class. The new class will serve as super class and others as subordinate classes (which derive attribute from the super clas

A super class is a class from which another class inherits properties super class shares its properties with its child classes. A subclass is a d that inherits attributes and methods from a super class.

Subclasses are not limited to the methods provided to them by the su class. They can also have their own data members and methods apart fr those inherited from the super class.

The process of combining the common attributes and functions am a group of related classes may be repeated at different levels of abstrac with the *sole* objective of extending classes. As hierarchy structure becom progressively higher, the amount of specification and implementation inher by the lower level classes increases. We may repeat the process until we sure that no new class can be formed. Fig. 2.1 shows two level iterat process.



(a) Objects in Solution Spaces *(b) First Level of Hierarchy* *(c) Second Level of Hierarchy*

Fig. 2.1

An example of super class and subclass is air tickets. Air tickets can be of two types, confirmed and request. Both these tickets have a lot of common attributes, for example, flight number, date, destination and time. However, a confirmed ticket would also have a seat number, while a request ticket would have a status.

In this example, air ticket is a super class and two subclasses are confirmed and request which inherits same properties or attribute from super class.

Q.33. What is nested class ? How does it differ from local class ? Create a date as a nested class of student qualifier class and do the manipulation of reading and printing the date of birth and date of joining of a student.

Ans. A class declared inside the declaration of another class is called ***nested class.*** Nested classes provide classes with non-global status. Host and nested classes follow the same access rules for members that exist between non-nested classes. Nested classes could be used to hide specialized classes and their instances within a host class.

Classes can be defined and used inside a function or a block. Such classes are known as local classes.

Problem –

```
#include <iostream.h>
class student
{
    private :
        int roll_no;
        char name[30];
        char branch[20];
        int marks;
    public :
        class date
```

```
        {
            int day;
            int month;
            int year;
        public :
            date( )
            {
                day = month = year = 0;
            }
        void set_dob( )
        {
            cout << "Enter date of birth–";
            cout <<"Enter day:";
            cin >> day;
            cout << "Enter month:";
            cin >> month;
            cout << "Enter year:";
            cin >> year;
        }
        void show_dob( )
        {
            cout << "Date of birth:" << day<< "–" month << "–"
                                                    year << endl;
        }
        void set_join( )
        {
            cout << "Enter date of joining–";
            cout << "Enter day:";
            cin >> day;
            cout << "Enter month:";
            cin >> month;
            cout << "Enter year:";
            cin >> year;
        }
        void show_join( )
        {
            cout << "Date of joining:" << day << "–" << month <<
                                                "–" << year << endl
        }
        } dob, yoj;
    student( )
    {
        roll_no = 0;
        name[0] = '\0';
        branch[0] = '\0';
        marks = 0;
    }
```



```
    ~student( )
    {
    }
    void set( )
    {
        cout<< "Enter student details–";
        cout << "Roll_no:";
        cin >> roll_no;
        cout << "Student_name:";
        cin >> name;
        cout << "Student_branch:";
        cin >> branch;
        cout << "Marks:";
        cin >> marks;
        dob.set_dob( );
        yoj.set_join( );
    }
    void show( )
    {
        cout << "Roll_no:" << roll_no;
        cout << "Student_name:" << name;
        cout << "Student_branch:" << branch;
        cout << "Marks:" << marks;
        dob.show_dob( );
        yoj.show_join( );
    }
};

int main( )
{
    student s1;
    s1.set( );
    s1.show( );
    return 0;
}
```

Q.34. What are the different types of object diagrams ?

Ans. There are two types of object diagrams –

(i) Class diagrams (ii) Instance diagrams.

A class diagram is a schema, pattern or template for describing many possible instances of data. A class diagram describes object class.

An instance diagram describes how a particular set of objects relate to each other. An instance diagram describes object instances. Instance diagrams are

MsInvent

useful for documenting test cases (specially scenarios) and discussing exam
A given class diagram corresponds to an infinite set of instance diagrams.

Fig. 2.2 shows a class diagram and one possible instance diagram desc by it.

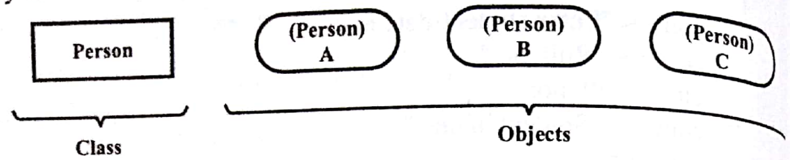


Fig. 2.2 Object Diagrams

The symbol for an object instance is a rounded box. The class name parentheses is at the top of the object box in boldface. Object names are li in normal font. The symbol for class is a box with class name in boldface.

Class diagrams describe the general case in modeling a system. Insta diagrams are used mainly to show examples to help to clarify a complex d diagram.

The distinction between the class diagrams and the instance diagrams in fact artificial, classes and instances can appear on the same object diag but in general it is not useful to mix classes and instances.

Q.35. What is the purpose of class diagram ? Describe icons used class relationship. ***(R.G.P.V., Dec. 2012, 20***

Ans. Once CRC cards and use cases have been generated, class diagr provide a way to express the relationships among the classes, and to rec the information on the CRC cards. The Universal Modeling Language (UM specifies how class relationships are depicted on class diagrams. Explor class relationships with the class diagrams may again reveal that addition changes need to be made to the original classes, resulting in further revisi to the CRC cards.

When the CRC cards, use cases, and class diagrams have been complet the final program design is ready to be coded.

Icons Used for Class Relationships – The essential connections amo classes are association, inheritance, "has", and "using" relationships, wh icons are shown in fig. 2.3. Each such relationship may include a textual la that documents the name of the relationship or suggests its purpose. Relations names need not be global, but must be unique within their context.

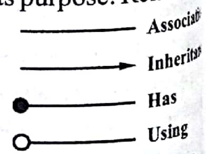


Fig. 2.3 Class Relationship Icon

The association icon connects two classes and denotes a semantic connection. Associations are often labeled with noun phrases, such as *Employment*, denoting the nature of the relationship. A class may have an association to itself (called a reflexive association). It is also possible to have more than one association between the same pair of classes. Associations may be further adorned with their cardinality using the syntax in the following examples –

0	None
1	One
*	Some (0 to infinity)
0..1	None or one
1..*	One or more
2..4	Two, three, or four
7, 11	Seven or eleven

The cardinality adornment is applied to the target end of an association, and denotes the number of links between each instance of the source class and instances of the target class.

The inheritance icon denotes a generalization/specialization relationship (i.e., the "is a" relationship), and appears as an association with an arrowhead. The arrowhead points to the superclass, and the opposite end of the association designates the subclass. According to the rules of the chosen implementation language, the subclass inherits the structure and behaviour of its superclass. Also according to these rules, a class may have one (single inheritance) or more (multiple inheritance) superclasses. Name clashes among the superclasses are also resolved according to the rules of the chosen language. In general, there may be no cycles among inheritance relationships. Also, inheritance relationships may not have cardinality adornments.

The "has" icon denotes a whole/part relationship (the "has a" relationship also known as aggregation), and appears as an association with a filled circle at the end denoting the aggregate. The class at the other end denotes the part whose instances are contained by the aggregate object.

The using icon denotes a client/supplier relationship, and appears as an association with an open circle at the end denoting the client. This relationship indicates that the client in some manner depends upon the supplier to provide certain services. It is used to indicate the decision that operations of the client class invoke operations of the supplier class, or have signatures whose return class or arguments are instances of the supplier class.

Q.36. What is the purpose of class diagram ? Describe icons used for class relationships. How we can use object as software modules ? (R.G.P.V., Dec. 2014)

Ans. Refer to Q.35 and Q.20.

Q.37. What do you mean by class diagram and object diagram ?

Ans. **Class Diagram –** A class diagram is used to show the existence of classes and their relationships in the logical view of a system. A single class diagram represents a view of the class structure of a system. During analysis, we use class diagrams to indicate the common roles and responsibilities of the

entities that provide the system's behaviour. During design, we use class dia to capture the structure of the classes that form the system's architecture.

The two essential elements of a class diagram are classes and their relationships.

Object Diagram – An object diagram is used to show the existence of objects and their relationships in the logical design of a system. A single object diagram represents a view of the object structure of a system. During analysis we use object diagrams to indicate the semantics of primary and secondary scenarios that provide a trace of the system's behaviour. During design use object diagrams to illustrate the semantics of mechanisms in the logical design of a system.

The two essential elements of an object diagram are objects and relationships.

Q.38. What is use of class diagram and object diagram ? Describe icons used for class relationship. How can we use objects as software mod

(R.G.P.V., June 2

Ans. Refer to Q.35, Q.37 and Q.20.

Q.39. What are the elements of a class diagram ? Explain with exam Discuss any two advanced concepts of class diagram and their significa

Ans. **Elements of a Class Diagram** – The two essential elements class diagram are classes and their basic relationships are as follows –

(i) Classes – Fig. 2.4 shows the icon used to represent a class class diagram. Its shape is that of a cloud. Some call it an amorphous bl

A name is required for each class. If the name is particularly long, it either be elided or the icon magnified. Every class name must be unique to its enclosing class category. For certain languages, most notably C++ and Smalltalk, we may further constrain these semantics to require that every class name be unique to the system.

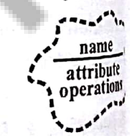


Fig. 2.4 Class I

For certain class diagrams, it is useful to expose some of the attributes operations associated with a class. Using the following language-independent sy an attribute may have a name, a class, or both, and optionally a default value

- A Attribute name only
- : C Attribute class only
- A : C Attribute name and class
- A : C = E Attribute name, class, and default expression

An attribute name must be unambiguous in the context of the class

An operation denotes some service provided by the class. Operations usually just named when shown inside a class icon, and are distingu

from attributes by appending parentheses or, where necessary for the purposes of the diagram, by providing the operation's complete signature –

- N( ) Operation name only
- R N(Arguments) Operation return class, name, and formal arguments (if any)

Operation names must be unambiguous in the context of the class, according to the rules for overloading in the chosen implementation language.

(ii) Class Relationships – Refer to Q.35.

Example – Fig. 2.5 shows an example of this notation, drawn from the problem of the hydroponics gardening system. This diagram describes only a small part of the hydroponics system class structure. Here we see the class *GardeningPlan,* which includes an attribute named *crop* together with one modifier operation, *execute*, and one selector operation, *canHarvest.* There is an association between this class and the class *EnvironmentController,* wherein instances of the plan define the climate that instances of the controller monitor and modify.

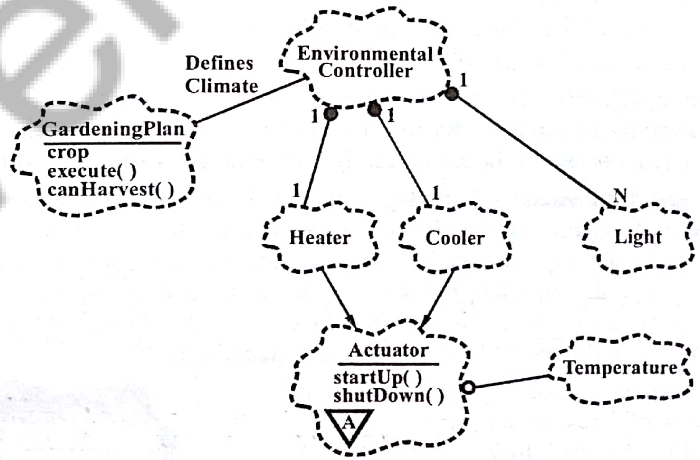


Fig. 2.5 Hydroponics Gardening System Class Diagram

This diagram also indicates that the class *EnvironmentController* is an aggregate whose instances contain exactly one heater, one cooler, and any number of lights. The *Heater* and *Cooler* classes in turn are both subclasses of the abstract class *Actuator*, which provides the protocol *startUp* and *shutDown*, and which uses the class *Temperature.*

Advanced Concepts of Class Diagram – The two advanced concepts of class diagram are as follows –

(i) Parameterized Classes – Some object-oriented programming languages, most notably C++, Eiffel, and Ada, provide parameterized classes. A parameterized class denotes a family of classes whose structure and behaviour are defined independent of its formal class parameters. We must match these

formal parameters with actual ones (the process of instantiation) to form a conc class in this family; by *concrete* class, we mean one that may have instances.

Parameterized classes are sufficiently different than plain classes to warra a special adornment. As the example in fig. 2.6 shows a parameterized class is visualized as a simple class, but with a dashed-line box in the upper right corner denoting its formal parameters. An instantiated class is adorned with a solid-line box denoting its actual parameters, matched positionally to the corresponding formal parameters. In either case, we may optionally supply the formal and actual parameters as text inside the box.

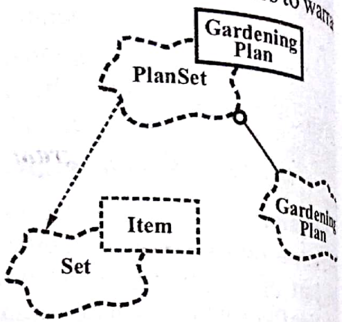


Fig. 2.6 Parameterized Classe

The instantiation relationship between a parameterized class and instantiated class is shown as a dashed line, pointing to the parameterized cla Most often, the instantiated class requires a "using" relation to other concre classes (such as *GardeningPlan* in this example) for use as an actual parame

A parameterized class may not have any instances and may not itself used as a parameter. An instantiated class defines a new class distinct from other concrete classes in the same family whose actual parameters differ.

(ii) Metaclasses – Languages such as Smalltalk and CLOS provi metaclasses. A metaclass is the class of a class. In Smalltalk, the most comm use of metaclasses is to provide class instance variables and operations, simi to the C++ practice of static members, or to define factory operations th generate instances of the corresponding class. In CLOS, metaclasses play important role in the ability to tailor that language's semantics

Metaclasses are also sufficiently different than plain classes to warrant a special adornment. As we show in fig. 2.7, a metaclass is visualized as a simple class, but with a gray-filled icon. The meta relationship is shown as a directed thick gray line, and points from a class to its metaclass. In this example, the metaclass provides the factory operations *new* and *default* for generating new instances of the class *GardeningPlan*.



Fig. 2.7 Metaclasses

A metaclass may not itself have any instances, but may inherit fro contain instances of, use, and otherwise associate with other classes.

Meta relationships have one other use. On certain class diagrams, it useful to show an object that serves as a static member for some class. show the class of this object, we may draw a meta relationship from the obje

to its class. This is consistent with the earlier use – a meta relationship shows the connection between some entity (either an object or a class) and its class.

Q.40. Write short note on metaclass. ***(R.G.P.V., June 2013)***

Or

What is metaclass ? ***(R.G.P.V., June 2015)***

Ans. Refer to Q.39.

Q.41. Draw a class diagram for online course reservation system. ***(R.G.P.V., June 2014)***

Ans.

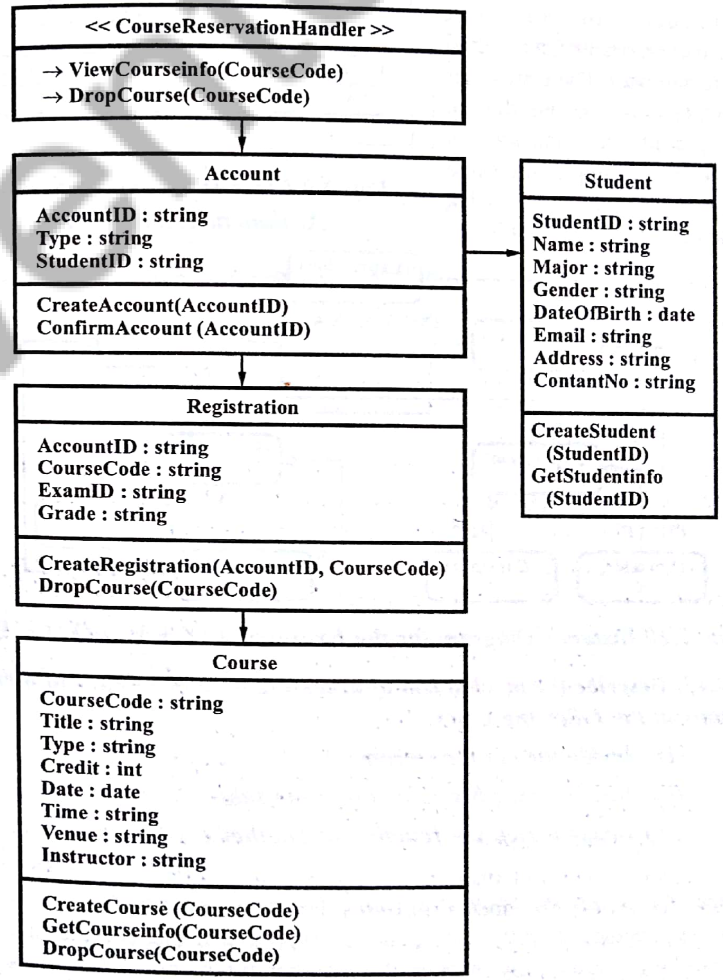


Fig. 2.8

Q.42. What are class diagrams ? How is it useful in object orien programming ? Take any one example to laborate class diagrams.

(R.G.P.V., Dec. 20

***Ans.* Class Diagram** – Refer to Q.37.

A class diagram for simple arithmetic expressions is shown in fig. 2.9

Fig. 2.10 shows an instance diagram for the expression (X + Y/3)/(X/4 + Y).

The object diagram in the above figure is actually a metamodel for binary expressions. Parentheses are needed for an infix representation but are not required in the metamodel. There are other representations, like postfix, in which parentheses also are not required. For example, the same expression in a postfix representation is XY3/+X4/Y+/.

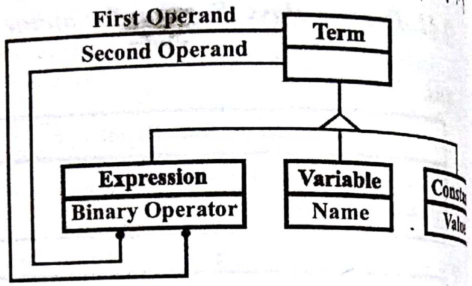


Fig. 2.9 Class Diagram for Simple Arithmetic Expressions

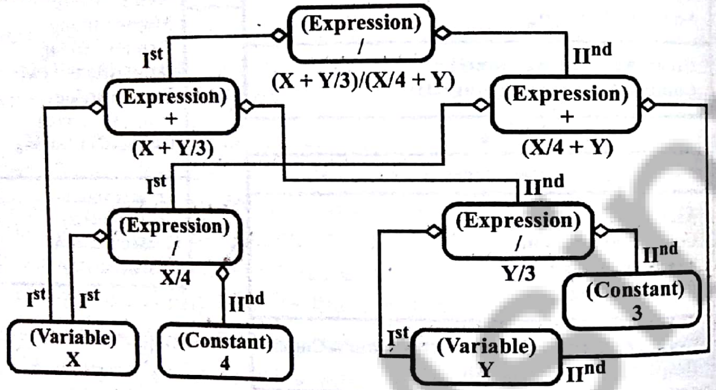


Fig. 2.10 Instance Diagram for the Expression (X + Y/3)/(X/4 + Y)

Q.43. Describe the mechanism of accessing data member and memb functions in the following cases –

(i) Inside the main program

(ii) Inside a member function of the same class

(iii) Inside a member function of another class.

(R.G.P.V., Dec. 20

Ans. (i) Inside the Main Program – The data members that are priv cannot be accessed directly, inside the main program. The private data of class can be accessed only through the member functions of that class. T data members and member functions that are declared public, can be accessed outside the class. For example –


```
class item
{
    int number;
    public:
     int x;
     void getdata( );
     void display( );
};
int main( )
{
    item i;
    i.number;                    // error, number is private
    i.x = 10;                    // ok, x is public
    i.getdata( );
    i.display( );
    return 0;
}
```

As shown in the above example, only public data members and member functions are directly accessed inside the main program. Private data members (like number) cannot be accessed directly inside the main program.

(ii) Inside a Member Function of the Same Class – Inside a member function of the same class, both public and private data members and member functions can be accessed. For example –

```
class example
{
    int x;
    public:
        void get( )
        {
         cout << "Enter value of x ";
         cin >> x;
         disp( );
        }
        void disp( )
        {
         cout << "Value is " << x;
        }
};
int main( )
{
    example e;
    e.get( );
    return 0;
}
```

Here in the above example, we find that get( ) function can access b private data x and public member function disp( ). Thus, inside the mem function of the same class we can access both private and public data memb as well as member functions.

(iii) Inside a Member Function of Another Class – We can acc public data members and public member functions, inside a member func of another class. We cannot access private data members and private mem function, inside a member function of another class. For example –

```
class example
{
        int x;
        public:
            void get( )
            {
              cout << "Enter value of x = ";
              cin >> x;
            }
            void disp( )
            {
              cout << endl;
              cout << "Value of x = " << x;
            }
        };
   class example1
        {
            int y;
            public:
        void get1( )
        {
          cout << "Enter value of y ";
          cin >> y;
        }
        void disp1( )
        {
            example e;
            e.get( );
            e.disp( );
            cout << endl;
            cout << "Value of y = " << y;
        }
};

int main( )
{
        example1 e1;
        e1.get1( );
        e1.disp1( );
        return 0;
}
```


Here in the above example, we find that public member function get( ) of class example is accessed in disp1( ) function of class example1. This shows that public data members and member functions can be accessed inside a member function of another class.

Q.44. Explain the term attributes and methods in class and object.

Or

Compare the attribute and method.

Ans. **Attributes** – Having decided what our abstract data type is going to represent, we have to identify what elements go to make up its state. These are known as ***attributes.*** When we identify attributes, we are not specifying values for them. For example, we might decide that our data type must have the attribute of ***colour***, but we do not predetermine what that colour might be for any individual coffee cup, only that all coffee cups will have a colour of some kind.

Methods – The behaviour of our 'thing' is defined by its methods (also called operations or services), which are processing routines related to the data type. In abstract data type, the method acts as the only path between the user and the state data. It is normal practice to limit access to the data so that it can only be accessed through these methods.

There are several types of methods. Two of them are selector method and modifier method.

A selector method is ***read*** method, one which allows access to an attribute, but do not allow that attribute to be altered. This is analogous to the idea of a function. It is also known as 'get' method. A 'modifier' method is a ***write*** method, allowing attributes to be altered. This is analogous to the idea of a procedure and is also known as a 'set' method.

Q.45. Discuss memory requirements for classes and objects.

Ans. On creating a class, memory is not allocated to its data member. Thus, there exists a template, but data members cannot be manipulated unless an instance of this class is created by defining an object. It might gives an impression that memory is allocated to both data member and member function of the class when an object of a particular class is created. It is partly true because memory is allocated only to data member not to the member function of the class when an object is created. Member functions are created and stored in memory only once when a class specification is declared. All objects of class access same memory area where the member functions are stored. The member functions for all the objects are same and there is no point in allocating a separate copy for each and every object created using the same class specification. However, an individual memory is allocated for every object's data members because they hold different values. It enables different objects to handle their data in a manner that suits them.

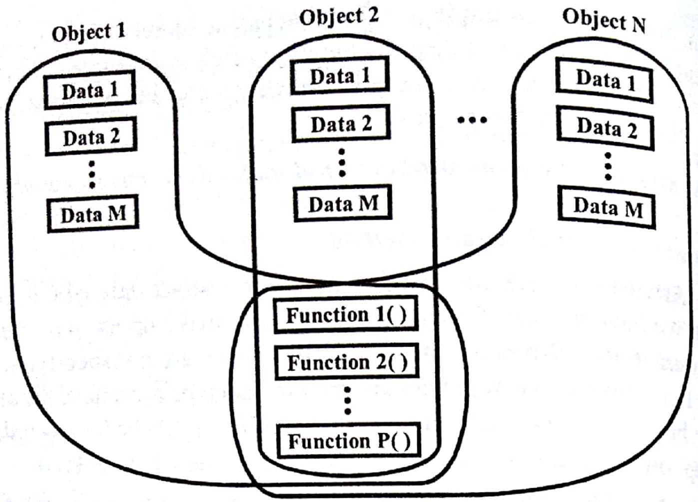


Fig. 2.11

Fig. 2.11 shows the organization of memory resource for the objects. In fig. 2.11, we see that N objects of the same class are created and data members of those objects are stored at different memory locations, while the member functions of N objects are stored at same memory location. Therefore, each object contains a separate copy of data members and different objects share the member function among them. It is simpler to visualize each object as containing both its own data and its own member functions. But in some situations, such as estimating the size of an executing program, it is helpful to know what is happening behind the scenes.

Q.46. Write a short note on arrays of object.

Ans. We can have array of variables that are of the type ***class***. Such variables are called ***arrays of objects.*** For example,

```
class employee
{
    char name [20];
    int age;
    public :
        void getdata (void);
        void putdata (void);
};
```

The identifier ***employee*** is a user-defined data type and can be used to create objects that relate to different categories of the employees. Example

```
employee manager [2];       //array of manager
employee foreman [10];      //array of foreman
employee worker [100];      //array of worker
```


The array ***manager*** contains two objects (managers) namely, manager [0] and manager [1]. Similarly, the ***foreman*** array contains 10 objects, (foreman) and the ***worker*** array contains 100 objects (workers).

Q.47. Describe arrays as class data member.

Ans. Arrays can be used as data items in classes. As an example, consider a program, stackarray that creates a simple stack class.

```
#include <iostream.h>
class stack
{
    private :
    enum {MAX = 10};
    int st[MAX];
    int top;
    public:
    stack( )
        {top = 0;}
    void push (int var)
    {st[++ top] = var;}
int pop( )
    {return st[top – –];}
};
int main( )
{
    stack 01;
    a1.push(01);
    a2.push(02);
    cout <<"1:"<<a1.pop( ) << endl;
    cout <<"2:"<<a1.push( ) << endl;
    a1.push(03);
    a1.push(04);
    a1.push(05);
    a1.push(06);
    cout<<"3:"<<a1.pop<<endl;
    cout<<"4:"<<a1.pop<<endl;
    cout<<"5:"<<a1.pop<<endl;
    cout<<"6:"<<a1.pop<<endl;
    return 0;
}
```

The main( ) program in stackarray exercise the stack class by creating an object a1 of the class. It push two items on the stack, and then pop them and display them. Thereafter, it push four items, pop them and display them. The stack class is an example of an important feature of object oriented programming using a class to implement a container or data storage mechanism.

Msinverter

Q.48. Write a program to explain how an object can be returned from function. **(R.G.P.V., June 20..)**

Ans. A function cannot only receive objects as arguments, but can also return them. That is, it is possible to return an object through the called function. Whenever a function returns an object, it is necessary that a function return type must be the class name of that object.

Example –

```
//Program for complex class
#include <iostream.h>
using namespace std;
class complex                          //x + iy form
{
      float x;                         // real part
      float y;                         //imaginary part
public:
      void input(float real, float imag)
      {   x = real; y = imag;}
      friend complex sum(complex, complex);
           void show(complex);
};
complex sum(complex c1, complex c2)
{
      complex c3;
      c3.x = c1.x + c2.x;
      c3.y = c1.y + c2.y;
      return(c3);
}
void complex : : show(complex c)
{
      cout << c.x << "+j" <<c.y << "\n";
}
int main( )
{
      complex a, b, c;
      a.input(3.1, 5.65);
      b.input(2.75, 1.2);
      c = sum (a, b);
      cout << "a ="; a.show(a);
      cout << "b ="; b.show(b);
      cout << "c =", c.show(c);
      return 0;
}
```

Output –

a = 3.1 + j 5.65, b = 2.75 + j1.2, c = 5.85 + j 6.85

MsInventer

Q.49. Differentiate between the following –
(i) Objects and classes
(ii) Encapsulation and abstraction
(iii) Class and metaclass.

(R.G.P.V., Dec. 2012)

Ans. (i) Object and Classes – The differences between objects and classes are as follows –

S.No.	Objects	Classes
(i)	A single object is simply an instance of a class.	A class is a set of objects.
(ii)	Objects may be created and destroyed at run time.	Once we have defined a class, it exists all the time a program is running.
(iii)	An object has identity, state and behaviour.	A class has a unique name, attributes and methods.
(iv)	For example, fruit mango; where mango is an object.	For example, fruit mango; where fruit is a class.

(ii) Encapsulation and Abstraction – The differences between data abstraction and data encapsulation are as follows –

S.No.	Encapsulation	Abstraction
(i)	It is wrapping up of data and functions into a single unit (called class).	It refers to the act of representing essential features without including the background details or explanations.
(ii)	It focuses upon the implementation that gives rise to this behaviour.	It focuses upon the observable behaviour of an object.
(iii)	It is a process of including in an object everything it needs, hidden from other objects in the system.	They encapsulate all the essential properties of the objects that are to be created.

(iii) Class and Metaclass – The class holds the attributes and methods which will apply to objects of the class – it is the class of the objects.

The metaclass holds the attributes and methods which will apply to the class itself – therefore it is the class of class.

Every class has only one metaclass, and the metaclass has those parts of a class which are not appropriate to be duplicated for every specific object. The metaclass is basically a repository for those parts of a class which must exist at the run time of a program, whether or not there are objects of the class in existence.

Q.50. What is the difference between class and object ? **(R.G.P.V., June 20**

Or

Differentiate between class and object. **(R.G.P.V., Dec. 20**

Ans. Refer to Q.49 (i).

Q.51. Explain the difference between structure and class.

Ans. The difference between structure and class are as follows –

S.No.	Structure	Class
(i)	A structure is a convenient tool for handling a group of logically related data items.	A class is a way to bind the da and its associated functio together.
(ii)	Structure holds only data.	Classes hold both the data a functions.
(iii)	The members of a structure are public.	The members of a class a private or public.
(iv)	The variables declared inside the structure are called structure members.	The variables declared insi the class are known as da members.

Q.52. What do you mean by access control specifiers ? Explain each them in brief.

Or

What are the visibility labels for members ? **(R.G.P.V., Dec. 20**

Or

Explain access control specifiers. **(R.G.P.V., Dec. 20**

Ans. When we define a class, we define members. A user can have diffe access privileges to the members. A class differentiates between acc privileges by dividing its contents and associating each one of them with one of the following keywords –

(i) Public (ii) Private (iii) Protected.

These keywords are known as visibility labels or access-control specifi If no keyword is specified, then the members are considered as private.

(i) Public – Public means all the class members declared un public will be available to everyone. The data member and member functi declared public can be accessed by other classes too. Hence, there are chan that they might change them. So the key members must not be declared pub

```
class XYZ
{
   .....
   public :
   int a;
   void display ( );
   ......
}
```


(ii) Private – Private means that no one can access the class members declared private outside that class. If someone tries to access the private member, they will get a compile time error. By default class variables and member functions are private.

```
class XYZ
{
   ......
   private :
   int a;
   void display ( );
   ......
}
```

(iii) Protected – Protected is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class.

```
class XYZ
{
   ......
   protected :
   int a;
   void display ( );
   ......
}
```

Q.53. Differentiate and give examples to bring out the difference between public and private access specifiers.

Ans. All data members and members functions of a class which are to be accessible outside the class should be declared in public section.

```
class Test
{
   ......
   public :          // Access specifier
   int age;          // Public data member
   int getage ( );   // Public member function
   ......
};
   Test t
   int a = t.age;    // can access public data
   t.getage ( );     // can access public function
```

The private members of a class have strict access control. All data members and member functions of the class are accessible only within the same class. The private members of a class are inaccessible outside the class.

```
class Test
{
    ......
        private :           // Access specifier
        int age;            // Private data member
        int getage ( );     // Private member function
        .......
};
        Test t
        int a = t.age;      // cannot access private data
        t.getage ( );       // cannot access private function
```

Q.54. In what respect are protected and private data member differen

Ans. The protected keyword is like private, in that the outside world c access class member in a protected section only by using class members. T difference between ***private*** and ***protected*** comes into play only within class derived from the base class. Members of a derived class can access protec members of the base class directly, but they cannot directly access priva members of the base class. So members in the protected category behave li private members as far as the outside world is concerned, but behave li public members as far as derived class is concerned.

Example –

```
#include <iostream.h>
using namespace std;
class B
{
        int a;          // private; not inheritable
protected:      // protected, ready for inheritance
        int b;
        void get_ab( );
        int get_a(void);
        void show_a(void);
};
class D : private B // private derivation
{
        int c;
public:
        void mul(void);
        void display(void);
};
void B : : get_ab(void)
```



```
{
    cout << "Enter values for a and b:";
    cin >> a >> b;
}
int B : : get_a( )
{
    return a;
}
void B : : show_a( )
{
    cout << "a = " << a << "\n";
}
void D : : mul( )
{
    get_ab( );
    c = b * get_a( ); // a cannot be used directly
}
void D : : display( )
{
    show_a( ); // outputs value of 'a'
    cout << "b =" << b << "\n";
    cout << "c =" << c << "\n";
}
int main( )
{
    D d;
  // d.get_ab( );    WON'T WORK
    d.mul( );
  // d.show_a( ); WON'T WORK
    d.display( );
  // d.b = 20; WON'T WORK; b has become private
    d.mul( );
    d.display( );
    return 0;
}
```

Output –

```
Enter values for a and b : 5 10
    a = 5
    b = 10
    c = 50
Enter values for a and b : 12 20
    a = 12
    b = 20
    c = 240
```

MsInventer

Q.55. *What are static data member and static member functions?*

Or

What do you mean by static class members? Explain the characteris of static class members with suitable examples. ***(R.G.P.V., Dec. 20***

Ans. **Static Data Member –** A data member of a class is qualified static.

The properties of a ***static*** member variable are same as that of a C sta variable. A static member variable contains certain special characteristics. Th are as follows –

(i) It is initialized to zero when the first object of its class is creat No other initialization is allowed.

(ii) Only one copy of that member is constructed for the wh class and is shared by all the objects of that class, no matter how ma objects are created.

(iii) It is visible only within the class, but its lifetime is the wh program.

Static variables allow us to maintain values common to the entire cl For instance, a static data member is used as a counter that records t occurrences of all the objects. Program given below illustrates the use o static data member –

```
#include <iostream.h>
using namespace std;
class item
{
    static int count;
    int number;
    public:
    void getdata(int a)
    {
        number = a;
        count++;
    }
    void getcount(void)
    {
        cout<<"count:";
        cout<<count<<"\n";
    }
};
int item : : count;
int main( )
```



```
{
    item a, b, c ;                      //count is initialized to zero
    a.getcount( );                      //display count
    b.getcount( );
    c.getcount( );
    a.getdata(100);                     //getting data into object a
    b.getdata(200);                     //getting data into object b
    c.getdata(300);                     //getting data into object c
    cout<<"After reading data"<<"\n";
    a.getcount( );                      //display count
    b.getcount( );
    c.getcount( );
    return 0;
}
```

Static Member Functions – Similarly, there are static member functions. A member function that is declared static has the following characteristics –

(i) A static function accesses only other static members (functions or variables) declared in the same class.

(ii) A static member function is called using the class name (instead of its objects) as follows –

class_name : : function_name;

The following program shows the implementation of these characteristics.

```
#include <iostream.h>
class test
{
    int code;
    static int count;       //static member variable
    public:
        void setcode( )
        {
            code = ++ count;
        }
        void showcode( )
        {
            cout <<"object number :"<< code <<"\n";
        }
        static void showcount(void)     //static member function
        {
            cout<<"count:" <<count<<"\n";
        }
};
int test : : count;
int main( )
{
    test t1, t2;
    t1.setcode( );
    t2.setcode( );
```

Msinverter

```
        test : : showcount( );
        test t3;
        t3.setcode( );
        test : : showcount( );
        t1.showcode( );
        t2.showcode( );
        t3.showcode( );
        return 0;
}
```

Q.56. When do we declare a member of class static ?

Ans. We declare a member of class as static, when all objects of the s class share a common item of information. A static data member is avail only within the class, but it continues to live till the time program execution d not come to an end. In that sense a static data member is similar to the ordin static variable. However, their utility is different. While a normal static variab used to retain information between calls to a function, static data members class are used to share information among the objects of a class.

Q.57. What is the difference between register variable and autom variable ? Explain with example. What are other storage classes variables ? **(R.G.P.V., June 20**

Ans. The extent of the variable is the period of time during which memory is associated with a variable. It is characterized by storage clas The storage class of a variable specifies the allocation of storage space to t variable by the compiler. The extent of a variable is defined by the stor classes. There are four types of storage classes in C++ as follows –

(i) Auto Variables – All the variables are defined as auto variab by default. They are created when the function/block is entered and destroy when the function/block is finished. For local auto variables the memory sp is allocated on the stack. The global auto variables are visible to all the modu of a program and therefore, they are not defined many times unlike declarations. For example –

```
#include <iostream.h>
int cube(int a);
int main( )
{
        auto int x = 6;
        cout <<"Cube of" << x <<" is : " << cube(x);
}
int cube(int a)
{
        auto int temp;
        temp = a * a * a
        return(temp);
}
```


(ii) Register Variables – The execution of a program is speeded up by the allocation of CPU (processor) register to variables. When such variables are accessed memory is not referred. The number of variables, which are declared as register are limited (typically two or three), within any function or as global variables. A program that uses register variables executes faster than a similar program without register variables. It is the responsibility of the compiler to allot register variables. These variables are treated as auto variables if the compiler is unable to do so. It is better to define frequently used variables like loop indices, as register variables. For example –

```
#include <iostream.h>
#include <string.h>
{
        char name[30];
        register int i;
        cout << "Enter a string";
        cin >> name;
        cout << "The reverse of the string is";
        for(i = strlen(name – 1); i > = 0; i – –)
        cout << name[i];
}
```

(iii) Static Variables – The static storage class is used to define a variable whose scope is limited to either a block, a function or a file and extent is the life-span of a program. The global heap allocates the memory space for local static and global variables. Static variables that are defined within a function retain their values from the previous call. For example,

```
#include <iostream.h>
void Print(void)
{
        static int count = 1;
        cout << "count = "<<count <<endl;
        count = count +1;
}
int main( )
{
        print( );
        print( );
}
```

The program outputs a sequence of numbers starting with 1, instead of a string of 1's. Although, these static variables cannot be accessed from other parts of the program.

(iv) Extern Variables – When a program spans across different files, they can share information using global variables. Global variables are defined only once in any of the program module and they are accessed by all others. It is obtained by declaring such variables as extern variables. It conveys the compiler that such variables are defined in some other file. For example, consider

a program having the following files –

```
//file1.cpp : module one defining global variables
int done // global variable definition
void func1( )
{
    .........
    .........
}
void disp( )
{
    .........
    .........
}
//file2.cpp : module two of the project
extern int done //global variable declaration
void func3
{
    .........
    .........
}
```

In file1.cpp, the statement

int done;

defines the variable done as a global variable. In file2.cpp, the statement

extern int done;

declares the variable done and specifies that it is defined in some other file is noted that the definition of the variable done must appear in any one of t modules, whereas extern declaration can appear in any or all modules of program. The linker binds all references to the same memory location whe encounters such variables. Thus, any changes to the variable done is visible all the modules accessing it.

Q.58. Explain object instantiation with an example.

Ans. In order to instantiate objects, we need to call on an abstract da type – a ***class*** in C++. As we know, each class has a unique name by which may be identified. Objects instantiated with unique names are those whe lifetimes and identities are predictable at compile time.

An object is instantiated in C++ by declaring its type, followed by name, just like other data types. For example,

```
int x;              // an instances of type int
char y;             // an instances of type char
BankAccount z;      // an instance of type 'BankAccount'
```

When we declare x to be of type integer, it means that we can treat x the same as any other integer, and perform integer arithmetic on it. By declar

z to be of type 'BankAccount', we treat z as any other BankAccount, which means it will have the attributes and methods defined by the class.

Q.59. Write a short note on object interactions. ***(R.G.P.V., Dec. 2003)***

Or

How objects of two different classes can interact ? Give an example for it. ***(R.G.P.V., June 2005, Dec. 2013)***

Ans. A request for execution of a procedure is a message for an object, and therefore will call a function in the receiving object that provides the desired result. Message passing includes specifying the name of the object, the name of the function (message) and the information to be sent. For instance –

Employee.Salary(name);

Object ↑ Message ↑ Information ↑

An object-oriented program made of a set of objects that communicate with each other. Therefore, the process of programming in an object-oriented language has the following steps –

(i) Creating classes that define objects and their behaviour,

(ii) Creating objects from class definitions, and

(iii) Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass message to one another. Objects contain a life cycle. They are constructed and destroyed. Communication with an object is feasible as long as it remains alive.

Objects in an object-oriented programming interact by sending message with one another. For instance, if there are "customer" and "account" in a program then the "customer" object sends a message to the account object requesting for bank balance. Customer object has data and code to manipulate the data. Customer object interacts with account object without having to know details of each other's data or code. Objects can interact without having to know details of each other's data or code. It is enough to know the type of message accepted and the type of response returned by the objects during object interactions.

Q.60. Describe how objects interact and how they are instantiated. ***(R.G.P.V., Dec. 2011)***

Or

Describe how objects are instantiated and how they interact with each other ? ***(R.G.P.V., Dec. 2012)***

Ans. Refer to Q.59 and Q.58.

Q.61. What is an object ? How object of two different classes can interact ? Give an example for it. **(R.G.P.V., June 20..)**

Ans. Refer to Q.8 and Q.59.

Q.62. What are the different forms of message passing ?

Ans. Message passing may take one of the following forms –

(i) Synchronous – An operation commences only when the sender has initiated the action and the receiver is ready to accept the message. The sender and receiver will wait indefinitely until both parties are ready to proceed.

(ii) Balking – It is same as synchronous, except that the sender will abandon the operation if the receiver is not immediately ready.

(iii) Time Out – It is same as synchronous, except that the sender will only wait for a specified amount of time for the receiver to be ready.

(iv) Asynchronous – A sender may initiate an action regardless of whether the receiver is expecting the message.

Q.63. What is message passing ? How objects are interacted with one another ? **(R.G.P.V., Nov. 20..)**

Ans. Refer to Q.59 and Q.632

Q.64. With the help of a complete C++ program illustrate the following
(i) Class declaration (ii) Object declaration
(iii) Message passing (iv) Object passing.

Ans. The following program illustrates the declaration of the class the declaration of objects with the operations on its objects and object passing

```
#include <iostream.h>
using namespace std;
class time                    //class declaration
{
    int hours;
    int minutes;
    public:
        void settime(int h, int m)
        {   hours = h; minutes = m;
        }
        void gettime(void)
        {
            cout < < hours << "hours and";
            cout << minutes << "minutes" << "\n";
        }
        void sum(time, time);
};
```



```
void time : : sum(time t1, time t2)
{
    minutes = t1.minutes + t2.minutes;
    hours = minutes/60;
    minutes = minutes % 60;
    hours = hours + t1.hours + t2.hours;
}
int main( )
{
    time T1, T2, T3;          //object declaration
    T1.gettime(3, 45);        //message passing
    T2.gettime(2, 30);        //message passing
    T3.sum(T1, T2);           //object passing
    cout << "T1 = ";          T1.gettime( );
    cout << "T2 = ";          T2.gettime( );
    cout << "T3 = ";          T3.gettime( );
    return 0;
}
```

Q.65. What is constructor ? How it is declared and defined ?

Ans. A constructor is a member function of the class. It is called each time when an object of that class is created. Therefore, when an object is created an initialization needs to be done which automatically done by the constructor function. It is known as constructor, since it constructs the value of data members of the class.

A constructor is declared and defined as follows –

```
class integer
{
        int m, n;
    public :
        integer(void); //constructor declared
        ..........
        ..........
};
integer : : integer(void) //constructor defined
{
    m = 0;
    n = 0;
}
```

A constructor function has same name as the class of which it is a part and has no return type in it.

When a class has a constructor, it ensures that an object created by the class will be initialized automatically. When an object of that class is created, it

initializes its data members. It is not required to call the constructor fun separately.

Whereas the normal member function is defined in the class and we to initialize that member data, then we need to call them separately. This w be inconvenient, if there are a large number of objects.

A ***default constructor*** is a constructor that accepts no parameters. default constructor of *class* A is A : : A( ).

If no default constructor is defined, then the compiler supplies a def constructor. Therefore, a statement

A a;

calls the default constructor of the compiler to create the object a.

Q.66. What is constructor ? Is it mandatory to use constructor in a cla

(R.G.P.V., Nov. 20

Ans. Refer to Q.65.

It is mandatory to use constructors to initialize instance variable i class. One the object for the class is created.

Q.67. Write the characteristics of constructor functions.

Ans. The constructor functions have some special characteristics as follow

(i) They should be declared in the public section.

(ii) They do not have return types, not even void and they can return values.

(iii) They are called automatically when the objects are created

(iv) Like other C++ functions, they can have default argument

(v) They cannot be inherited, though a derived class can call t base class constructor.

(vi) They cannot be referred by their addresses.

(vii) Constructors cannot be *virtual*.

(viii) They make 'implicit calls' to the operators *new and delete* w memory allocation is required.

Q.68. What is constructor ? List the properties of constructor functi

(R.G.P.V., June 20

Ans. Refer to Q.65 and Q.67.

Q.69. What is parameterized constructor ? Explain.

Ans. The constructor can also be used to initialize the data members of the objects to zero. Although, it may be necessary to initialize the different d elements of different objects with the different values when they are create C++ allows us to get this by passing parameters to the constructor functi when the objects are created. The constructors that take arguments are know as ***parameterized constructors.***

MSInventer

Example –

```
class integer
{
        int m, n;
    public :
        integer(int x, int y); // parameterized constructor
        ..................
        ..................
};
integer : : integer(int x, int y)
{
    m = x;
    n = y;
}
```

When a constructor has been parameterized, the object declaration of the statement like

integer in1;

may not work. For this we will have to pass initial values as arguments to the constructor function when an object is declared. That is,

integer in1 = integer(0, 100); //explicit call

or

integer in1 (0, 100); //implicit call

It is that, when the constructor is parameterized, the appropriate arguments for the constructor must be provided.

Q.70. Distinguish between the following two statements –

Time T2(T1);

Time T2 = T1;

T1 and T2 are the objects of class time. ***(R.G.P.V., Dec. 2005)***

Ans. A copy constructor is used to declare and initialize an object from another object. For example, the statement

Time T2(T1);

would define the object T2 and at the same time initialize it to the values of T1. Another form of this statement is

Time T2 = T1;

The process of initializing through a copy constructor is known as ***copy initialization.*** Here the statement T2 = T1; will not invoke the copy constructor. However, if T1 and T2 are objects, this statement is legal and simply assigns the values of T1 to T2, member-by-member. This is the task of the overloaded assignment operator (=).

Q.71. Write a short note on destructors. ***(R.G.P.V., Nov. 2018)***

Ans. The complement of the constructors are the destructors. The destructor function has the same name as the constructor, but it is preceded

by a ~ (tilde sign). The objective of a destructor is to kill the object and to in releasing the memory occupied by the object. That is

```
~ integer( )
{ }
```

A destructor never takes any argument nor does it return any value. invoked implicity by the compiler upon exit from the program (or block function) to free the storage that is no longer accessible.

Q.72. What is global constructor and destructor ? (R.G.P.V., Dec. 20

Or

Explain constructor and destructor with examples. (R.G.P.V., Dec. 20

Ans. **Constructor** – Refer to Q.65.

Destructor – Refer to Q.71.

Example –

```
#include <iostream.h>
class class_name
{
        int a, b;
    public:
        class_name(int x, int y); //Constructor
        ~ class_name( ); //destructor function
        void show( );
};
class_name : : class_name(int x, int y)
{
    a = x;
    b = y;
}
void class_name : : show( )
{
    cout <<"\n a =" <<a<<" b =" << b;
}
class_name : : ~class_name( )
{
    cout << "\n Destructing the object :";
}
int main( )
{
class_name obj(6, 8);
obj.show( );
return 0;
}
```

The output will be

a = 6 b = 8

Destructing the object :

Q.73. What is meant by copy constructor ? What type of operations will cause the copy constructor to be invoked ? Write a program to display the use of copy constructor.

Or

Explain what type of operation causes to call copy constructor to be invoked. Write an example to show the use of copy constructor. (R.G.P.V., Dec. 2015)

Ans. A copy constructor takes a reference to an object of the same class as itself as an argument. A reference variable has been used as an argument to the copy constructor. We cannot pass the argument by value to a copy constructor. When no copy constructor is defined the compiler supplies its own copy constructor. The copy constructor allows us to instantiate an object as an exact copy of another in terms of its attributes values.

Definition invokes a copy constructor – A constructor that creates a new object and copies its argument into it. The default copy constructor which is provided automatically by the compiler for every object, performs a member-by-member copy.

The copy constructor is invoked when an object is passed by value to a function. It creates the copy that the function operates on. Thus, if the function

```
void func (alpha);
```

were declared and this function were called by the statement

```
func (a1);
```

then the copy constructor would be invoked to create a copy of the a1 object for use by func( ).

Example –

```
#include <iostream.h>
using namespace std;
class code
{
        int id;
    public:
        code( ) { }                  // constructor
        code(int a) {id = a;}        // constructor
        code(code &x)                // copy constructor
        {
            id = x.id;               // copy in the value
        }
        void display(void)
        {
            cout << id;
        }
};
```

```
int main( )
{
        code A (100);
        code B (A);
        code C = A;
        code D;
        D = A;
        cout << "\n id of A : " ; A.display( );
        cout << "\n id of B : " ; B.display( );
        cout << "\n id of C : " ; C.display( );
        cout << "\n id of D : " ; D.display( );
        return 0;
}
```

The default copy constructor can be used on any objects without b explicitly defined, though it is possible to provide a user defined c constructor if additional processing is required, or if the default behaviour i appropriate, perhaps because the objects contain pointer. Since the def copy constructor copies pointer address rather than the data they refere we may have, in some cases, to override this behaviour.

Q.74. What is the importance of copy constructor ? Under wh condition explicit definition of copy constructor is necessary ? Explain proper examples. ***(R.G.P.V., June 2***

Ans. The importance of copy constructor is described below. Let's be by restating the problem that, the copy constructor is designed to solve. default when one object is used to initialize another, C++ performs a bitw copy, that is, an identical copy of the initializing object is created in the ta object. Although this is perfectly adequate for many cases (and generally exa . what you want to happen) there are situations in which a bitwise copy sho not be used. One of the most common is when an object allocates mem when it is created. For example, assume a class called *Myclass* that alloca memory for each object when it is created, and an object A of that class. T means that A has already allocated its memory. Further, assume that A is us to initialize B, as shown here –

Myclass B = A;

If a bitwise copy is performed, then B will be an exact copy of A. T means that B will be using the same piece of allocated memory that A is usi instead of allocating its own. Clearly, this is not the desired outcome. F example, if *Myclass* includes a destructor that frees the memory, then t same piece of memory will be freed twice when A and B are destroyed.

The same type of problem can occur in two additional ways – first, wh a copy of an object is made when it is passed as an argument to a functio second, when a temporary object is created as a return value from a functio

Remember, temporary objects are automatically created to hold the return value of a function and they may also be created in certain other circumstances.

To solve the type of problem just described, C++ allows you to create a ***copy constructor,*** which the compiler uses when one object initializes another. When a copy constructor exists, the default bitwise copy is bypassed.

Following is an example where an explicit copy constructor function is needed. This program creates a very limited "safe" integer array type that prevents array boundaries from being overrun. Storage for each array is allocated by the use of **new**, and a pointer to the memory is maintained within each array object –

```
#include <iostream.h>
#include <new.h>
#include <cstdlib.h>
using namespace std;
class array {
        int * p;
        int size;
public:
        array(int sz) {
        try {
                p = new int[sz];
        } catch(bad_alloc xa) {
          cout<< "Allocation failure\n";
          exit(EXIT_FAILURE);
        }
        size = sz;
     }
  ~array( ) {delete[ ] p ;}
  // copy constructor
  array(const array &a);
  void put(int i, int j) {
  if(i > = 0 && i < size) p[i] = j; }
  int get(int i) {return p[i]; }
  };
  // copy constructor
  array : : array(const array &a) {
  int i;
  try {
  p = new int[a.size];
  } catch(bad_alloc xa) {
     cout<< "Allocation Failure \n";
     exit(EXIT_FAILURE);
  }
  for(i = 0; i < a.size; i++) p[i] = a.p[i];
}
```

```
int main( )
{
      array num[10];
      int i ;
      for(i = 0; i < 10; i++) num.put(i, i);
      for(i = 9; i >= 0; i– –) cout << num.get(i);
      cout<< "\n";
      // create another array and initialize with num
      array x(num) ; // invokes copy constructor
      for(i = 0; i < 10; i++) cout << x.get(i);
      return 0;
}
```

Now look closely at what happens when **num** is used to initialize x in statement –

array x(num); // invokes copy constructor

The copy constructor is called, memory for the new array is allocated and stored in **x.p**, and the contents of **num** are copied to **x's** array. In this way, x and **num** have arrays that contain the same values, but each array is separate and distinct. (That is, **num.p** and **x.p** do not point to the same piece of memory.) If the copy constructor had not been created, the default bitwise initialization would have resulted in **x** and **num** sharing the same memory for their arrays. (That is, **num.p** and **x.p** would have indeed pointed to the same location).

Notice that the copy constructor is called only for initializations. For example, this sequence does not call the copy constructor defined in the above program –

```
array a(10);
// . . .
array b(10);
b = a;      // does not call copy constructor
```

In this case, **b = a** performs the assignment operation. If '=' is not overloaded (as it is not here), a bitwise copy will be made. Therefore, in some cases, you may need to overload the = operator as well as create a copy constructor to avoid certain types of problems.

Q.75. Explain the use of dynamic constructor with example.

Ans. The constructors can be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus, resulting in the saving of memory. Allocation of memory to objects at the time of their construction is called the dynamic construction of objects. The memory is allocated with the help of the new operator.

AsInventer

Example –

```
#include <iostream.h>
#include <string.h>
using namespace std;
class string
{
        char *name;
        int length;
    public :
        string( )                       // constructor
        {
            length = 0;
            name = new char[length + 1];
        }
        string(char *s)                 // constructor
        {
            length = strlen(s);
            name = new char[length + 1];
                                        // one additional character for '\0'
            strcpy(name,s);
        }
        void display(void)
        {
            cout << name << "\n";
        }
        void join(string &a, string &b);
};
        void string : : join(string &a, string &b)
        {
            length = a.length + b.length;
            delete name;
            name = new char[length + 1];
                                        // dynamic allocation
            strcpy(name, a.name);
            strcat(name, b.name);
        };
        int main( )
        {
            char *first = "Rahul";
            string name1(first), name2("Kumar"), s1;
            s1.join(name1, name2)
            name1.display( );
            name2.display( );
            s1.display( );
            return 0;
        }
```

Output –

```
Rahul
Kumar
Rahul Kumar
```

Q.76. In which order are the constructors and destructors called, an object of the derived class is created explain with program.

Or

Write a program in C++ to show how constructors are implem... when the classes are inherited. **(R.G.P.V., Dec. 2...)**

Ans. The constructors play an important role in initializing objects... derived class need not have a constructor as long as the base class has a... argument constructor. However, if the base class has constructors with... or more arguments, then it is necessary for the derived class to have... constructor and pass the arguments to the base class constructors. In... application of inheritance, objects of the derived class are created. Hence... makes sense for the derived class to pass arguments to the constructor of... base class. When an object of a derived class is created, the constructor of... base class is executed first and then the constructor of the derived class.

Unlike constructors, destructors in the class hierarchy are invoked in... reverse order of the constructor invocation. The destructor of that class whose constructor was executed last, while building object of the derived class,... be executed first. If destructors are missing in any class in the hierarchy... classes, that class's destructor is not invoked. The following program sh... the order of invocation of constructors and destructors in handling insta... of a derived class –

```
//order of invocation of constructors and destructors
#include <iostream.h>
class base1                    //base class
{
    public:
      base1( )
      {
        cout <<"\n No argument constructor of the base class bas...
      }
      ~base1( )
      {
        cout <<"\n Destructor in the base class base1";
      }
};
class base2                    //base class
```



```
{
    public :
        base2( )
        {
            cout<<"\n No argument constructor of the base class
                                                          base2";
        }
        ~base2( )
        {
            cout<<"\n Destructor in the base class base2";
        }
};
class derived : public base1, public base2
{
    public :
        derived( )
        {
            cout<<"\n No argument constructor of the derived
                                                   class derived";
        }
        ~derived( )
        {
            cout <<"\n Destructor in the derived class derived";
        }
};
int main( )
{
    derived d;
    retrun 0;
}
```

Output –

```
No argument constructor of the base class base1
No argument constructor of the base class base2
No argument constructor of the derived class derived
Destructor in the derived class derived
Destructor in the base class base2
Destructor in the base class base1
```

Q.77. Can a constructor be private ? If so what it means. How to use as a private ?

Ans. Yes, a constructor can be private. A private constructor taking no arguments and having no other constructors. A private constructor is a special

instance constructor. It is commonly used in classes that contain s members only. In such a case, other classes except nested classes are allowed to create instances of this class. For example –

```
class Myclass
{
    private Myclass( )          //private constructor
    { }
    public static double e = 2.71828;
}
```

The declaration of the empty constructor prevents the automatic genera of a default constructor. Note that if you do not use an access modifier w the constructor it will still be private by default. However, the private modi is usually used explicitly to make it clear that the class cannot be instantia The following is an example of a class using a private constructor –

```
using System;
public class Myclass
{
    private Myclass( )
        { }
    public static int counter;
    public static int IncCount( )
    {
        return ++counter;
    }
}
class MainClass
{
    static void main( )
    {
        //If you comment the following statement, it will generate
        //an error because the constructor is inaccessible
        //Myclass myobject = new MyClass( );      //Error
        Myclass.counter = 100;
        Myclass.IncCount( );
        Console. writeLine ("New count : {0}", Myclass.counter);
    }
}
```

UNIT 3

RELATIONSHIPS – INHERITANCE – PURPOSE AND ITS TYPES, 'IS A' RELATIONSHIP

Q.1. Define the term inheritance with an example.
(R.G.P.V., June 2016)

Or

What does inheritance mean ? ***(R.G.P.V., Dec. 2003)***

Ans. Inheritance is the process by which objects of one class acquire the properties of objects of another class, i.e., inheritance is the process of creating new classes called *derived classes,* from the existing classes, called *base classes.*

The derived classes inherit all the properties of the base classes, but can add their own properties. In this process, the properties of base classes are unchanged.

The general form of defining a derived class is as follows –

```
class derived_class : visibility_mode base_class
{
    .......... // members of derived class
    ..........
};
```

Here, the colon indicates that derived_class is derived from the base_class. The visibility_mode is optional, and if it present, may be either *private* or *public.* The default visibility mode is *private.*

For example,

```
class ABC : private XYZ // private derivation
{
    members of ABC
};
class ABC : public XYZ   // public derivation
{
    members of ABC
};
```

Q.2. In what way inheritance is useful in OOPS ?

Or

Discuss the role of inheritance in object oriented programming.

(R.G.P.V., June 2

Ans. Inheritance is a natural way to model the world or a domain of discourse, and so provides a natural model for object-oriented analysis (O object-oriented design (OOD) and even object-oriented programming (O This is common in the artificial intelligence (AI) domain, where semantic use inheritance to understand the world by using classes and concepts generalization and categorization by reducing the real-world's inherent comple

Inheritance also provides for code and structural reuse. In the comp class all routines and structure available in class computer are available t subclasses. This kind of reuse takes advantage of the is-a-kind-of relation Class libraries also allow reuse between applications, potentially allowing or of-magnitude increases in productivity and reductions in defect rates, as lib classes have already been tested and further use provides further tes providing even greater reliability.

With differential programming a class does not have to be modified if close to what's required; a derived class can be created to specialize it. This av code redundancy, since code would have to be copied and modified otherwi

Q.3. How do you achieve multiple inheritance in a single inherita system ?

(R.G.P.V., Dec. 2

Ans. Single inheritance means that each class has only a single supercl One benefit of using a single inheritance is the absence of ambiguity. One w of achieving the benefits of multiple inheritance in a language with si inheritance is to inherit from the most appropriate class and add an obje another class as an attribute or aggregation. Therefore, as class designer, have two ways to borrow existing functionality in a class. One is to inheri and the other is to use the instance of the class as an attribute.

Q.4. What are the different types of inheritance ? Explain with exam

(R.G.P.V., June 2

Ans. There are following forms of the inheritance –

(i) Single Inheritance – A derived class with only one base cl is called single inheritance. That is,

It shows B is a base class and D is a derived class.

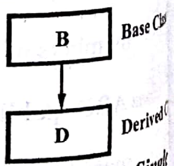


Fig. 3.1 Single Inheritance

The general form of defining it as follows –

```
class derived_class : visibility_mode base_class
{
    .......
    .......
};
```


(ii) Multiple Inheritance – A derived class with several base classes is called the multiple inheritance. That is,

It allows to combine the features of B1 and B2 in the derived class D with new ones.

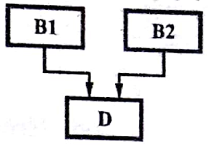


Fig. 3.2 Multiple Inheritance

The syntax of multiple classes is as follows –

```
class D : visibility B1, visibility B2
{
    .......
    ....... // (Body of class D)
};
```

(iii) Hierarchical Inheritance – The features of one class may be inherited by more than one class. It is called the hierarchical inheritance. That is,

Inheritance can be used to modify a class when it did not satisfy the requirements of a particular problem. Additional members are added through inheritance to extend the capabilities of a class.

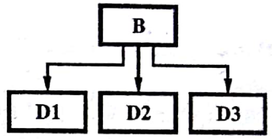


Fig. 3.3 Hierarchical Inheritance

In C++, such problem can be easily converted into class hierarchy. The base class includes all the features that are common to the subclasses. A subclass can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes.

(iv) Multilevel Inheritance – The mechanism of deriving a class from another "derived class", is called the multilevel inheritance.

Here, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. The class B is known as intermediate base class, since it provides a link for the inheritance between A and C. The chain ABC is known as inheritance path.

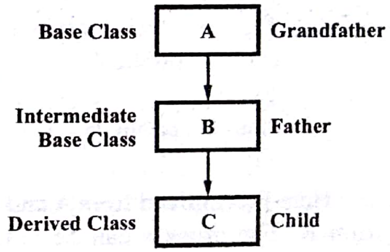


Fig. 3.4 Multilevel Inheritance

The derived class with multilevel inheritance is declared as follows –

```
class A                  // Base class
{
    .......
};
class B : public A       // B derived from A
```

```
{
    ........
};
class C : public B        // C derived from B
{
    .........
};
```

(v) Hybrid Inheritance – When we need two or more types of inheritance to design a program. This process is called hybrid inheritance.

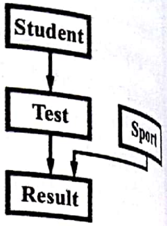


Fig. 3.5 Hybrid Inheri

Q.5. What is inheritance ? What are the different forms of inherita Write suitable codes to illustrate them. *(R.G.P.V., Nov.*

Or

What is inheritance ? What are the different types of inheritan Explain. *(R.G.P.V., Dec.*

Ans. Refer to Q.1 and Q.4.

Q.6. What is inheritance ? What are different levels of inheritance? Di the role of inheritance in reusability of software. *(R.G.P.V., Dec.*

Ans. **Inheritance** – Refer to Q.1.

Levels of Inheritance – Classes can be derived from classes that are themselves derived. For instance,

```
class A
{};
class B : public A
{};
class C : public B
{};
```

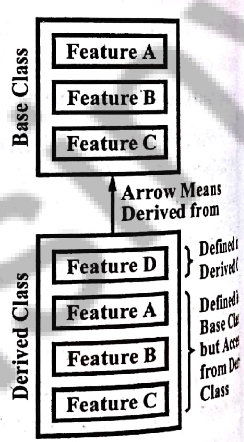


Fig. 3.6 Inheritanc

Here B is derived from A and C is derived from B. The process can be extended to an arbitrary number of levels, i.e., D could be derived from C, and so on.

Role of Inheritance in Reusability of Software – Inheritance important part of OOP. Its big payoff is that it permits code reusa Once a base class is written and debugged, it need not be touched a but, using inheritance, can nevertheless be adapted to work in diffe situations. Reusing existing code saves time and money and increa program's reusability. Inheritance can also help in the original conceptuali of a programming problem, and in the overall design of the program.

Q.7. Discuss the role of inheritance in object oriented programming. What is public and private derivation ? *(R.G.P.V., June 2011)*

Ans. **Role of Inheritance** – Refer to Q.2.

A derived class enhances its features by inheriting the properties of base class and adding features of its own. The declaration of a derived class describes its relationship with the base class in addition to its own features. The syntax of a derived class declaration is as follows –

```
class DerivedClass : [VisibilityMode] BaseClass
{
    .........
    .........
}
```

The colon (:) indicates the derivation of DerivedClass from the BaseClass. The VisibilityMode enclosed within the square brackets specifies that it is optional. The default visibility mode is private. The visibility mode must be either public or private. It means that the features of the base class are publicly or privately inherited.

If the base class is inherited publicly then the public members of the base class become public members of the derived class and the protected member of the base class become protected members of the derived class. Member functions and objects of the derived class treat these derived members as if they are defined in the derived class itself. We know that the public members of a class can be accessed by the objects of the class. Hence, the public members of the base class that are inherited publicly can be accessed by the objects of a derived class using the dot operator. But, protected members cannot be accessed with the dot operator.

If the base class is inherited privately then the public members of the base class become private members of the derived class and the protected members of the base class become private members of the derived class. Member functions and objects of a derived class treat these derived members as if they are defined in the derived class with the private modifier. Therefore, objects of a derived class cannot access these members.

Q.8. Describe multilevel inheritance with example. *(R.G.P.V., Dec. 2015)*

Ans. **Multilevel Inheritance** – Refer to Q.4 (iv).

Example –

```
#include <iostream.h>
class stud
```

```
{
    protected:
        rollno;
    public:
        void getno(void);
        void putno(void);
};
void stud : : getno(int a)
{
    rollno = a;
}
void stud : : putno( )
{
    cout << "Roll Number" << rollno<< "\n";
}
class test : public stud // first level
{
    protected :
        float sub1;
        float sub2;
    public :
        void getmarks(float, float);
        void putmarks(void);
};
void test : : getmarks(float a, float b)
{
    sub1 = a;
    sub2 = b;
}
void test : : putmarks( )
{
    cout << "Marks in subject1 =" <<sub1<< "\n";
    cout << "Marks in subject2 =" <<sub2 << "\n";
}
class result : public test // second level
{
        float total ; // private by default
    public :
        void show(void);
};
void result : : show(void)
{
    total = sub1 + sub2;
    putno( );
    putmarks( );
    cout << "Total =" << total << "\n";
}
int main( )
```



```
{
    result stud1;        //Create object
    stud1.getno(1001);
    stud1.getmarks(75.5, 56.5);
    stud1.show( );
    return 0;
}
```

Q.9. Define the following terms and give an example to show its realization in C++.

(i) Encapsulation *(ii) Class variables and class functions*
(iii) Repeated inheritance *(iv) Overloading.*

(R.G.P.V., Dec. 2009, June 2011)

Ans. *(i)* ***Encapsulation*** – Refer to Q.1 (Unit-II).

(ii) ***Class Variables and Class Functions*** – Refer to Q.31 (Unit-II).

(iii) ***Repeated Inheritance*** – Multiple inheritance brings up the possibility for a class to appear as a parent more than once in a class graph (repeated inheritance), and there is then a potential to share that class. C++ provides an alternative, where only parents specified as virtual (virtual bases) are shared within the same class lattice, allowing both shared and non-shared occurrences of a parent to coexist. All "features" in Eiffel (C++ members) of a repeated parent that are not to be shared must be renamed "along an inheritance path", else they are shared by default. This allows a finer granularity of control and consistent name resolution but requires more work for parents with many features.

With repeated inheritance, a subclass object may have multiple embedded objects for the same superclass. Fig. 3.7 shows that if both C and D inherit a class A, then an E object has two embedded A objects, one for C and the other for D. Class name A is not sufficient to identify the two A objects.

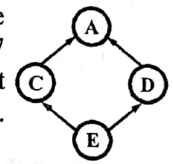


Fig. 3.7

(iv) ***Overloading*** – Refer to Q.25 (Unit-IV).

Q.10. Write a short note on inherited methods. ***(R.G.P.V., Dec. 2003)***

Ans. The ability to redefine a method name in a derived class allows us to tailor ***inherited methods*** for use with a derived class object. Inappropriate inherited methods may be overridden or extended by the derived class.

Object-orientation allows us to override any inherited method by defining a derived class method with the same name. We also have the facility, if required, to call any inherited methods within this new method so that the functionality of the existing implementation can be re-used and extended.

Inheritance means that all the methods and variables from the ancestor classes become methods and variables of the child class. Any fresh methods or variables of the child class in the hierarchy are simply added to the inherited

methods and variables. Any methods redefined at a child in the hiera override the inherited method.

When we override an inherited method in a subclass, we can increa access but not decrease it. Otherwise, we would destroy the ability of a sub object to behave like its superclass. In other words, a subclass can pro more functionality, but not less. Likewise a subclass can require less of methods parameters, but not more.

By overriding, an inherited method can be redefined. If an object recei message that does not have a method for that message in the class definition object works its way up in the hierarchy until it finds a method with that m

When a Java class inherits a method from its superclass, the bytec implementing method is not reproduced in the implementation of the cl When inheritance is used for specialization, a class extends its supercla order to redefine some of the superclass's methods, while leaving uncha the rest of the methods it inherits. The inherited methods that are redefined said to be overridden by the subclass. By overriding certain methods subclass specializes the behaviour of its superclass.

Q.11. Write a short note on multiple inheritance.

(R.G.P.V., June 2004, Dec. 2

Or

What is multiple inheritance ? When do we use such an inheritan Use an example to give the syntax of it. ***(R.G.P.V., Dec. 2***

Or

What is multiple inheritance ? Explain giving example.

(R.G.P.V., June 2

Or

What is multiple inheritance ? Explain its importance. Write a cod show the concept of multiple inheritance. ***(R.G.P.V., Dec. 2***

Ans. When a class is derived from more than one base class. This pro is called ***multiple inheritance.***

Multiple inheritance allows us to combine the features of several exis classes as a starting point for deriving new classes. It is like a child inheri the physical features of one parent and the intelligence of the other. In fig a class D is derived from base classes B1, B2 and B3.

The syntax for multiple inheritance is similar to that for single inherit In the situation shown in fig. 3.8 the relationship is expressed like this –

```
Class B1        // base class 1
{ .........
};
Class B2        // base class 2
{ .........
};
Class B3        // base class 3
{ .........
};
                // D is derived from B1, B2 and B3
Class D : public B1, public B2, public B3
{ .........
};
```

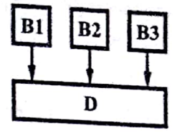



Fig. 3.8 Multiple Inheritance

Multiple inheritance does not mean that multiple subclasses share the same superclass. It also does not mean that a subclass can inherit from a class which itself is a subclass of another class. It is complex than single inheritance. But, it is very useful in certain circumstances.

Multiple inheritance is not supported by all object-oriented languages. It is supported by C++. It is complicated and not frequently used. Java does not provide multiple inheritance, at least not in the same sense C++ does. Like protected, multiple inheritance seems like a good idea but we know we need it only when we are face to face with right design problem. Since Java uses a single rooted hierarchy, we will probably run into fewer situations where multiple inheritance is necessary. The aforementioned interface keyword takes care of combining multiple interfaces.

Example –

```
#include <iostream.h>
class M
{
    protected :
        int m;
    public :
        void get_m(int);
};
class N
{
    protected :
        int n;
    public :
        void get_n(int);
};
class P : public M, public N
{
    public :
        void show(void);
};
void M : : get_m(int x)
```

```
{
    m = x;
}
void N : : get_n(int y)
{
    n = y;
}
void P : : show(void)
{
    cout << " m = " << m << " \n " ;
    cout << " n = " n << " \n " ;
    cout << " m * n = " << m * n << "\n" ;
}
int main( )
{
    P p;
    p.get_m(10);
    p.get_n(20);
    p.show( );
    return 0;
}
```

Q.12. Write a program to demonstrate multiple inheritance. ***(R.G.P.V., Dec.)***

Ans. Refer to Q.11.

Q.13. Write a program to implement multiple inheritance. Give advantages. ***(R.G.P.V., June)***

Ans. **Program** – Refer to Q.11.

Advantages of Multiple Inheritance – There are several advantages of multiple inheritance are as follows –

(i) Multiple inheritance permit a class to have more than one super class and to inherit features from all parents.

(ii) This permits mixing of information from two or more sources.

(iii) It has greater power in specifying classes and an increased opportunity for reuse.

Q.14. Explain the disadvantages of multiple inheritance. ***(R.G.P.V., June)***

Ans. Multiple inheritance can introduce new problems for the programmer. The two chief problems are inheriting different methods of the same name from two different base classes and inheriting multiple instances of a class via two or more related immediate base classes. Meeting these problems involves introducing a few new rules and syntax variations. Thus, using multiple inheritance is more difficult and problem-prone than using single inheritance. For this reason many in C++ community object strongly to multiple inheritance.

Q.15. How in the C++ programming language, virtual inheritance is a kind of inheritance that solves some of the problems caused by multiple inheritance ? ***(R.G.P.V., June 2015)***

Ans. Virtual inheritance is a type of inheritance in which the implementation of a superclass is incomplete, and a subclass is required for complete definition of an object. This type of inheritance can be used in conjunction with both single and multiple inheritance, but is most commonly used in multiple inheritance. Any class that inherits from a virtual base class becomes a direct subclass of that base class. A virtual base class may rely on a subclass to implement all its methods, but this is not a requirement.

C++ is the most commonly known computer language to use virtual inheritance. To declare virtual inheritance in C++, the virtual keyword is used. Both the superclass and subclass must declare virtual methods with the virtual keyword. This tells the C++ compiler that the superclass is incomplete, and it must get information from the subclass to complete it. Using the subclass to complete the superclass does not mean that subclasses overwrite each other if they have the same base class, and instead the C++ compiler takes care of determining what pieces go with each object. Since a virtual base class is required for virtual inheritance, global functions in C++ cannot be declared virtual. This inheritance type can only be used when adhering to object-oriented programming principles. The reason for this is that global functions are not associated with a particular class, and so are usually self-contained on their own. Without a superclass and subclass inheritance cannot happen, so global functions and virtual inheritance are mutually exclusive. Global functions can, in theory, be used inside virtual functions, but the reverse may not always work.

Virtual inheritance is used to solve many programming problems, and one of the most useful is resolution of ambiguity. In multiple inheritance, one may have a base class A that has two subclasses, B and C, and then a class D that inherits from both classes B and C. This pattern is commonly called the diamond of death because if classes A, B and C all have implementations of the same method, it is not possible for class D to determine which implementation it should use. Virtual inheritance solves this problem because each class's implementation remains distinct, and therefore unambiguous. This distinction is handled by specialized internal objects called virtual tables that keep track of each object type, but these tables do not need to be explicitly manipulated by a programmer because they are built into the language.

Q.16. Explain virtual inheritance in C++. ***(R.G.P.V., Dec. 2003)***

Or

Write a short note on disinheritance. ***(R.G.P.V., Dec. 2003, 2005)***

Or

What do you mean by the term 'disinheritance'. ***(R.G.P.V., Dec. 2006)***

Or

Explain disinheritance with example. ***(R.G.P.V., Dec. 2012, 2013)***

Ans. A programming language cannot - and should not - directly su... every concept from every design method. Similarly, a design language sh... not support every feature of every programming language. A design lang... should be richer and less concerned with details than a language suitab... systems programming must be.

When a programming language does not provide facilities for represen... a concept from the design directly, a conventional mapping between design... programming language construct should be used. For example, the design... specify that every operation not defined for a class A should be serviced b... object of a class B pointed to by a pointer p. C++ cannot express this dire... However, the expression of that idea ***(Disinheritance)*** in C++ is so stylized... one could easily imagine a program generating the code. Consider –

```
class B
{ – –
    void f( );
    void g( );
    void h( );
};
class A
{
    B * p;
    void f( );
    void ff( );
};
```

A specification that A ***disinherited*** to B through A : : p would resu... code like this –

```
Class A
{   B * p // virtual inheritance through p
    void f( );
    void ff( );
    void g( ) {p –> g( );} // disinherit g( )
    void h( ) {p –> h( );} // disinherit h( )
};
```

It is fairly obvious to a programmer what is going on here, but simula... this design concept of disinheritance in code is clearly inferior. Such relation... are not well understood by programming language and are less amenable... manipulation by tools.

Thus, it is clear that the concepts of ***disinheritance*** or ***virtual inherita...*** are totally theoretical concepts which are used when a true superclass/subcl... relationship does not occur.

Inheritance should only be used when a true generalization relationship occurs. Inheritance means that each instance of a subclass truly is an instance of the superclass. Improper use of inheritance leads to programs that are hard to maintain and extend.

Q.17. Discuss the following –
(i) Disinheritance (ii) Multiple inheritance.
(R.G.P.V., June 2012)

Ans. (i) Disinheritance – Refer to Q.16.
(ii) Multiple Inheritance – Refer to Q.11.

Q.18. How can base class protect derived classes so that changes to the base class will not affect them ? ***(R.G.P.V., Dec. 2004)***

Ans. One of the ways, by which a base class protects the derived classes so that changes to the base class does not affect them is by distinguishing public and private operations as –

Public operations are visible outside a class and have published interfaces. Once a public operation is used by other classes, it is costly to change its interface, so public operation should be carefully defined and change prone operations should be defined as private so that they will not be inherited.

Private operations are internal to a class and are used to help implement the public operations. Private operations can be deleted or their interfaces can be changed to modify the implementation of the class with impact limited to other methods on the class. Private methods add modularity. Internal details of the method only affect methods on the class, not other methods.

Q.19. Is it anyway possible for the objects of a derived class to access the private members of the base class ? If yes, How ?

Or

How can we make private members inheritable ? Explain in brief.

Or

Why are protected members said to be "inheritance ready" ?
(R.G.P.V., Dec. 2003)

Ans. Yes, the object of a derived class can access the private members of the base class by replacing the 'private' keyword in the base class with an alternative keyword 'protected'.

C++ provides a third visibility, protected, which serves a limited purpose in inheritance. A member declared as protected is accessible by the member functions within its class and any class derived immediately from it. The protected members are not accessible from outside the base or other derived classes of this base class.

When protected member is inherited in public mode, it becomes protected in the derived class too and, therefore, is accessible by the member functions of the derived class. It is also ready for further inheritance and hence protected

members are also said to be ***"inheritance ready"***. A protected member inh... in the private mode derivation becomes private in the derived class. Alth... it is available to the member functions of the derived class but it is not ava... for further inheritance. In protected derivation, both the public and prote... members of the base class becomes protected members of the derived c...

Therefore protected members are called inheritance ready becau... any mode they can be inherited at least once i.e., by the class derived immedi... from the base class containing the protected member.

Q.20. When deriving a class from a base class with protected inherit... public members of the base class become protected members of the de... class ? ***(R.G.P.V., June 20...)***

Ans. Protected inheritance is a variation on private inheritance. It us... keyword protected when listing a base class. With protected inheritance, pu... and protected members of a base class become protected members of the der... class. As with private inheritance, the interface for the base class is availab... the derived class, but not to the outside world. The main difference betw... private and protected inheritance occurs when you derive another class f... the derived class. With private inheritance, this third-generation class does... get the internal use of the base class interface. That's because the public b... class methods become private in the derived class, and private members... methods cannot be directly accessed by the next level of derivation. With prote... inheritance, public base methods become protected in the second genera... and so are available internally to the next level of derivation.

Q.21. How the ambiguity arises in multiple inheritance ?

Ans. In some cases, multiple inheritance is a straight-forward ama... mation of two or more different classes into a single class. The new cla... therefore, composed of other classes so that the whole is the sum of its p... This simplicity of structure is not always present, because there may be elem... of the inherited base classes which overlap in some way. This may be bec... the base classes have polymorphic methods. A derived class cannot in... more than one method with the same name without the programmer provi... some classification of which implementation of the method is being used... particular context. There may also be a common ancestor in the hiera... tree. Which means that the same attribute or method is inherited more t... once via different paths. Because of these possibilities, there are a numbe... circumstances in which multiple inheritance can cause ambiguity.

Ambiguity can arise when –

(i) A derived class inherits two methods with the same name... different implementation from its multiple base classes. These may or may... be semantically different.

(ii) A derived class overrides a multiple-derived inherited method, but calls a base class method as part of its implementation.

(iii) A derived class accesses an attribute or method inherited from a single ancestor by multiple base classes.

Q.22. What is basic properties of the public and private inheritance ? Explain disinheritance. ***(R.G.P.V., June 2015)***

Ans. When a base class is ***privately*** inheritated by a derived class, 'public members' of the base class become 'private members' of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.

On the other hand, when the base class is ***publicly*** inherited, 'public members' of the base class become 'public members' of the derived class and therefore they are accessible to the objects of the derived class. In both the cases, the private members are not inherited and, thus, the private members of a base class will never become the members of its derived class.

Disinheritance – Refer to Q.16.

Q.23. How ambiguity can be resolved ?

Ans. Two base classes may have member function with the same name. The name of the function alone is insufficient, since the compiler cannot figure out which of the two functions is meant. Consider the following two classes –

```
class M
{
    public:
        void display(void)
        {
            cout << "class M\n";
        }
};
class  N
{
    public :
        void display(void)
        {
            cout << "class N\n";
        }
};
```

Which display( ) function is used by the derived class when we inherit these two classes ? We can solve this problem by defining a named instance within the derived class, using the class resolution operator with the function

as shown below –

```
class P : public M, public N
{
    public :
        void display(void)   // overrides display( ) of M and N
        {
          M :: display( );
        }
};
```

We can now use the derived class as follows –

```
int main( )
{
    P p;
    p.display( );
    return 0;
}
```

Ambiguity may also arise in single inheritance applications. For ins[tance,] consider the following situation –

```
class A
{
    public :
        void display( )
        {
          cout <<"A\n";
        }
};
class B : public A
{
    public :
        void display( )
        {
          cout <<"B\n";
        }
};
```

In this case, the function in the derived class overrides the inh[erited] function and, therefore, a simple call to display( ) by B type object will [invoke the] function defined in B only. However, we may invoke the function define[d in A] by using the scope resolution operator to specify the class.

Example –

```
int main( )
{
    B b;
    b.display( );
    b.A::display( );
    b.B::display( );
    return 0;
}
```

Another kind of ambiguity arises if we derive a class from two classes that are each derived from the same class. This creates a diamond-shaped inheritance tree.

Example –

```
Class A
{
    public:
        void func( );
};
class B : public A
{ };
class C : public A
{ };
class D : public B, public C
{ };
int main( )
{
    D objD;
    objD.func( ); // ambiguous : won't compile
    return 0;
}
```

Classes B and C are both derived from class A, and class D is derived by multiple inheritance from both B and C. Trouble starts if we try to access a member function in class A from an object of class D.

In this example, objD tries to access func( ). However, both B and C contain a copy of func( ), inherited from A. The compiler cannot decide which copy to use, and signals an error.

There are various advanced ways of coping with this problem, but the fact that such ambiguities can arise causes many experts to recommend avoiding multiple inheritance altogether. We should certainly not use it in serious program unless we have considerable experience.

MsInventer

Q.24. When does ambiguity arise in multiple inheritance ? Ho resolve it ? Develop a program in C++ to create a derived class havi following items : Name, age, roll no, marks, empcode and designation. a base class student having data members as roll no, marks and the o base class is employee having data members as empcode and design These both base classes are inherited from a single base class person wi members name and age. The program should carry out the required in and output( ) member functions for all. **(R.G.P.V., June**

Or

What are the ambiguities that arise in multiple inheritance ? How they be removed ? **(R.G.P.V., June**

Ans. Refer to Q.21 and Q.23.

Program –

```
#include <iostream.h>
   class Person
   {
       protected:
       char name[20];
       int age;
       public:
       void InputPersonData( )
       {
          cout << "Enter Data for Person";
          cout << "Enter Your Name <19-Max>:";
          cin >> name;
          cout << "Enter Your Age:";
          cin >> age;
       }
       void OutputPersonData( )
       {
          cout << "Name:" <<name << endl;
          cout << "Age:" <<age<<endl;
       }
   };
       class Student : : virtual public Person
       {
          protected:
          int roll no, marks;
          public:
          void InputStudentData( )
          {
             cout << "Enter Data for Student";
             cout << "Enter Roll No.:";
             cin >> roll no;
             cout << "Enter the marks <max-100>:";
             cin >> marks;
          }
```



```
       void OutStudentData( )
       {
       cout<<"Roll No. :"<<roll no<<endl;
       cout<<"Marks:"<<marks<<endl;
       }
};
class Employee : : virtual public Person
{
    protected:
    int empcode;
    char desig[20];
    public:
    void InputEmployeeData( )
    {
    cout <<"Enter data for Employee";
    cout <<"Enter Employee code:";
    cin>>empcode;
    cout<<"Enter Employee Designation:";
    cin >> desig;
    }
    void OutputEmployeeData( )
    {
    cout <<"Empcode:"<<empcode<<endl;
    cout<<"Designation:"<<desig<<endl;
    }
};
class Detail : : public Student, public Employee
{
    public:
    void InputDetail( )
    {
        InputPersonData( );
        InputStudentData( );
        InputEmployeeData( );
    }
    void OutputDetail( )
    {
        OutputPersonData( );
        OutputStudentData( );
        OutputEmployeeData( );
    }
};
int main( )
{
        Detail d;
        cout << "INPUT";
        d.InputDetail( );
        cout << "OUTPUT";
        d.OutputDetail( );
        return 0;
    }
```

Q.25. What are virtual base classes ? How it overcome the disadvantage of multiple inheritance ? **(R.G.P.V., Dec. 2011)**

Or

Write short note on virtual base class.(R.G.P.V., Dec. 2001, June 200...)

Ans. Virtual base classes are mainly designed in order to overcome the disadvantage of multiple inheritance. Multiple inheritance refers to creating a new/derive class having more than one base classes. Under multiple inheritance concept, there are certain situations in which a derived class inherits the same protected data/variables from different base classes.

Consider a situation where all the three kinds of inheritance, namely multilevel, multiple and hierarchical inheritance, are involved. This is illustrated in fig. 3.9. The 'Child' has two direct base classes 'Parent1' and 'Parent2' which themselves have a common base class 'Grandparent'.

In this arrangement, a problem can arise if a member function in the **child** class want to access data or function in the **grandparent** class. The program shows what happens.

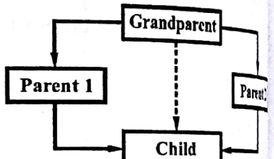


Fig. 3.9 Multipath Inheritance

```
// ambiguous reference to base class
class grandparent
    {
        protected:
            int basedata;
    };
    class parent1 : public grandparent
        {   };
    class parent2 : public grandparent
        {   };
    class child : public parent1, public parent2
        {
            public:
                int getdata( )
                { return basedata;}  // ERROR:
                                        ambiguous
        };
```

A compiler error occurs when the getdata( ) member function in child attempts to access basedata in **grandparent**. Why? when the **parent1** and **parent2** classes are derived from **grandparent**, each inherits a copy of **grandparent**, this copy is called a subject. Each of the two subjects contains its own copy of **grandparent's** data, including **basedata**. Now, when child refers to basedata, which of the two copies will it access? The situation is ambiguous and that's what the compiler reports.

To eliminate this ambiguity, we make parent1 and parent2 into virtual base classes, as shown by the following example –

```
// virtual base classes
class grandparent
    {
    protected:
        int basedata;
    };
class parent1: virtual public grandparent
                    // shares copy of grandparent
    {   };
class parent2 : virtual public grandparent
                    // shares copy of grandparent
    {   };
class child : public parent1, public parent2
    {
        public:
            int getdata( )
            { return basedata;} //ok: only one copy of grandparent
    };
```

Q.26. What is multiple inheritance and what are its associated problems ? How does the concept of inheritance of specifications help to overcome these problems ? Explain.

Ans. **Multiple Inheritance** – Refer to Q.11.

Difficulties in Multiple Inheritance – Conceptually, multiple inheritance is simple but it does introduce some practical problems for programming languages. Two issues must be addressed by languages – clashes among names from different superclasses, and repeated inheritance. Clashes will happen when more than one superclasses provide a field or operation with the same name or signature as a peer superclass. In C++, such clashes are resolved with explicit qualification. Repeated inheritance happens when more than one peer superclasses share a common superclass. In such a situation, inheritance lattice will be diamond-shaped, and so the question arises, does the leaf class have one copy or multiple copies of the structure of the shared superclass ? Some languages prevent repeated inheritance, some unilaterally choose one approach, and others such as C++, allow the programmer to decide. In C++, virtual base classes are used to denote a sharing of repeated structures, while non-virtual base classes result in duplicate copies appearing in the subclass.

Q.27. Discuss the various workarounds to deal with lack of multiple inheritance.

Ans. Dealing with lack of multiple inheritance is really an implementation issue, but early restructuring of a model is often the easiest way to work around its absence. Some restructuring techniques are described below. Two of the following

approaches make use of delegation, which is an implementation mechanism in which an object forwards an operation to another object for execution.

(i) Delegation using Aggregation of Roles – A superclass with multiple independent generalizations can be recast as an aggregate in which each component replaces a generalization. This approach is similar to that for accidental multiple inheritance. This approach replaces a single object having a unique ID by a group of related objects that compose an extended object. Inheritance of operations across the aggregation is not automatic. They may be caught by the join class and delegated to the appropriate component.

In this approach, the various join classes need not actually be created as explicit classes. All combinations of subclasses from the different generalizations are possible.

(ii) Inherit the Most Important Class and Delegate the Rest – Fig. 3.10 makes a join class a subclass of its most important superclass.

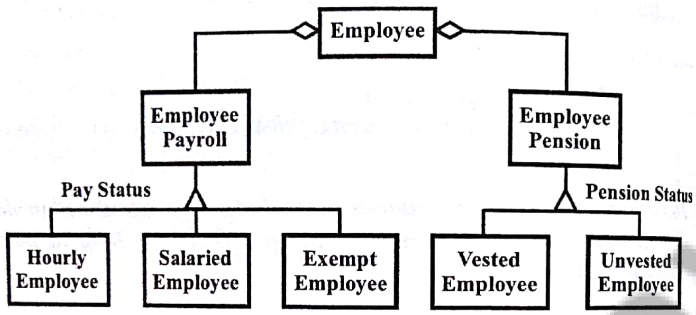


Fig. 3.10 Multiple Inheritance using Delegation

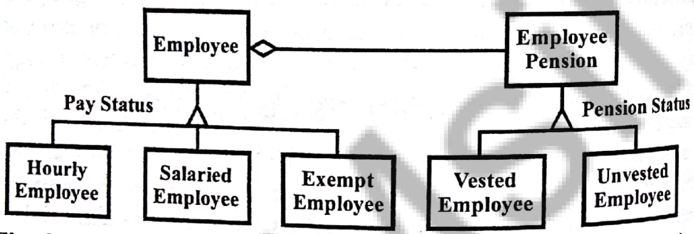


Fig. 3.11 Multiple Inheritance using Inheritance and Delegation

The join class is treated as an aggregation of the remaining superclasses and their operations are delegated as in the previous alternatives. This approach preserves identity and inheritance across one generalization.

(iii) Nested Generalization – Factor on one generalization first then the other. This approach multiplies out all possible combinations. For example, in fig. 3.12 under each of HourlyEmployee, SalariedEmployee, and ExemptEmployee, add two subclasses for vested and unvested employees. This preserves inheritance but duplicates declarations and code and violates the spirit of object-oriented programming.

Any of these workarounds can be made to work, but all compromise logical structure and maintainability. Some issues to consider when selecting the best workaround are –

(i) If a subclass has several superclasses, all of equal importance, it may be best to use delegation and preserve symmetry in the model.

(ii) If one superclass clearly dominates and the others are less important implementing multiple inheritance via single inheritance and delegation may be best.

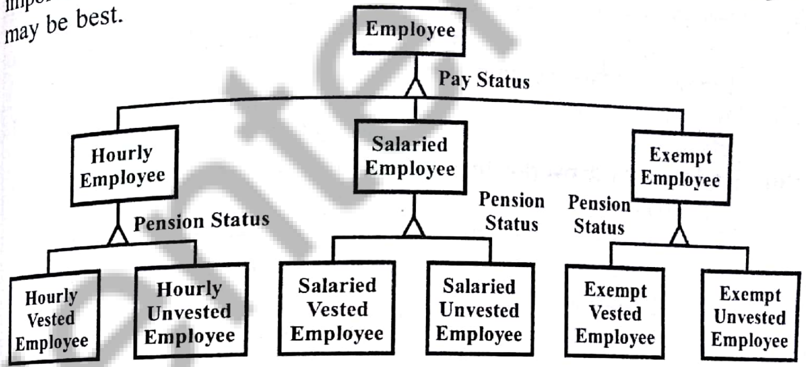


Fig. 3.12 Multiple Inheritance using Nested Generalization

(iii) If the number of combinations is small, consider nested generalization. If the number of combinations is large, avoid it.

(iv) If one superclass has significantly more features than the other superclasses or one superclass clearly is the performance bottleneck, preserve inheritance through this path.

(v) If you choose to use nested generalization, factor on the most important criterion first, the next most important second and so forth.

(vi) Try to avoid nested generalization if large quantities of code must be duplicated.

(vii) Consider the importance of maintaining strict identity. Only nested generalization preserves this.

Q.28. Differentiate between generalization and inheritance.

Ans. Generalization and inheritance are powerful abstractions for sharing similarities among classes while preserving their differences.

Inheritance refers to the mechanism of sharing attributes and operations using the generalization concepts.

Generalization is the relationship between a class and one or more refined versions of it. The class being refined is called the superclass and each refined version is called a subclass. For example, equipment is the superclass of pump and tank.

Q.29. Distinguish between the following terms –
(i) Inheritance and polymorphism
(ii) Dynamic binding and message passing.

Ans.(i) Inheritance and Polymorphism – The differences betw... inheritance and polymorphism are as follows –

S.No.	Inheritance	Polymorphism
(i)	Inheritance is the process in which new class reusing the properties of the existing ones.	Polymorphism simply means one name having multiple forms.
(ii)	It uses base class and derived class.	It does not use base class and derived class.
(iii)	It does not use overloading functions and operators.	It's used overloaded functions and operators.
(iv)	There are five types of inheritance – single, multiple, multi-level, hybrid and hierarchical.	There are two types of polymorphism – compile time and run time polymorphism.
(v)	It provides the idea of reusability.	It does not provide the idea of reusability

(ii) Dynamic Binding and Message Passing – The difference between dynamic binding and message passing are as follows –

S.No.	Dynamic Binding	Message Passing
(i)	Binding refers to the linking of a procedure call to the code.	Message passing does not refer to the linking.
(ii)	In dynamic binding, information does not pass from one computer to another.	Information should be passed from one computer to other.
(iii)	It is associated with polymorphism and inheritance.	It does not associate with polymorphism and inheritance

Q.30. Explain multipath inheritance by showing its example.

Ans. When a class derived from two or more classes, those are derived from the same base class, such a type of inheritance is known as multipath inheritance. The multipath inheritance also consists of many types of inheritance, such as multiple, multilevel, and hierarchical, as shown in fig. 3.13.

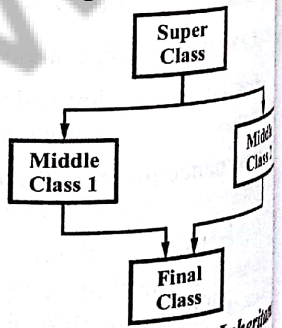


Fig. 3.13 Multipath Inheritance

In the above diagram a class named final derives properties from two classes named middle class 1 and middle class 2. These classes, in turn, derive properties from a single class named super. It implies that there will be two copies of class super in the final class. A super class may contain data members such as name and salary. Two names are fine but what about two salaries. What if you are an employer and not an employee.

Q.31. Write a program to show that the declaration of a class 'rectangle', which derives from the class 'square', which in turn derives from the class 'shape'.

Or

Partition a software development problem of your choice into classes, subclasses, objects and method at the highest level of design.

(R.G.P.V., June 2016)

Ans.
```
#include <iostream.h>
class shape
{
  public :
      void getdata( )
      {
          cout <<"Enter details –";
      }
};
class square : public shape
{
  private :
      int a, r;
  public :
  void getdata( )
  {
      shape : : getdata( );
      cout <<"Enter side of a square :";
      cin >> a;
  }
  void showdata( )
  {
      r = a*a;
      cout <<"Area of square is : "<<a<<endl;
  }
};
class rectangle : public square
{
  private
      l, b, r;
  public :
  void getdata( )
```

```
{
    square : : getdata( );
    cout <<"Enter length of rectangle:";
    cin >> l;
    cout <<"Enter breadth of a rectangle :";
    cin >> b;
}
void showdata( )
{
    square : : showdata( );
    r = l*b;
    cout <<"Area of rectangle is : "<< r << endl;
}
};
int main( )
{
    rectangle obj;
    obj.getdata( );
    obj.showdata( );
    return 0;
}
```

Q.32. Create a class called dimension containing their float data and a constructor to accept values. Also declare a pure virtual function area in it. ***(R.G.P.V., Dec. 201***

Or

Create a class called dimension containing three float data and constructor to accept values. Also declare a pure virtual function area() in it. Create three derived classes rectangle, square and triangle each inheriting dimension as public. Define corresponding constructors and redefine virtual function area in each. Write a complete program and also print the version of virtual function used each time area is printed. ***(R.G.P.V., Dec. 200***

Ans.

```
#include <iostream.h>
#include <conio.h>
class dimension
{
    protected:
        float l, b, h;
    public:
    dimension (float l1, float b1, float h1)
        {l = l1; b = b1; h = h1;}
        virtual void area( ) = 0;
};
```



```
class rectangle: public dimension
{
    float ar;
    public:
    rectangle (float l1, float b1, float h1): dimension (l1, b1, h1) { }
    void area( )
    {
        ar = l * b;
        cout <<"Virtual function of rectangle is called"<<"\n";
        cout <<"Area of rectangle: "<<ar<< "\n";
    }
};
class square: public dimension
{
    float ar;
    public :
    square(float l1, float b1 = 0, float h1 = 0) : dimension (l1, b1, h1)
    { }
    void area( )
    {
        ar = l * l;
        cout <<"Virtual function of square is called "<< "\n";
        cout << "Area of square is:" <<ar<< "\n";
    }
};
class triangle : public dimension
{
float ar;
public:
triangle (float l1, float b1, float h1): dim (l1, b1, h1) { }
    void area( )
    {
        ar = (b * h)/2;
        cout <<"Virtual function of triangle is called"<< "\n";
        cout << "Area of triangle is" <<ar<< "\n";
    }
};
```

```
int main( )
{
    clrscr( );
    dimension* dim[3];
    rectangle r1(2, 3);
    dim[0] = &r1;
    dim[0] -> area( );
    square s1(4.0);
    dim[1] = &s1
    dim[1] -> area( );
    triangle t1(0, 2, 4);
    dim[2] = &t1;
    dim[2] -> area( );
    getch( );
    return 0;
}
```

***Q.33. Design three class student, exam and results, where resul inherited from exam and exam is inherited from student. Write possi constructors to initialize the values. Write a main function to test t constructor execution by creating objects.* (R.G.P.V., June 2009, Dec. 20**

Ans.

```
#include <iostream.h>
class student
{
    protected:
    int rollno;
    public:
        void getno(int);
        void putno(void);
        student( ) {rollno = 0}
};
    void student : : getno(int a)
    {
        rollno = a;
    }
    void student : : putno( )
    {
        cout <<"Roll Number"<<rollno<<"\n";
    }
    class exam : public student        //First level inheritance
    {
        protected :
            float sub1;
            float sub2;
        public:
            void getmarks (float, float);
            void putmarks (void);
            exam( ) {sub1 = 0; sub2 = 0;}
};
```



```
void exam : : getmarks (float a, float b)
{
    sub1 = a;
    sub2 = b;
}
void exam : : putmarks( )
{
cout <<"Marks in subject1 =" <<sub1<<"\n";
cout <<"Marks in subject2 =" <<sub2<<"\n";
}
class results : public exam        //Second level inheritance
{
    float total ;
    public :
    void show(void);
    results( ) {total = 0}
};
void result : : show(void)
{
    total = sub1 + sub2;
    putno( );
    putmarks( );
    cout<<"Total="<<total<<"\n";
}
int main
{
result s1;
s1.getno(1001);
s1.getmarks(75.5, 56.5);
s1.show( );
return 0;
}
```

Q.34. Write a program which reads the bio-data and records of two sessional and then print final result. Make separate class for bio-data, sessional 1, sessional 2 and final result. Sessional 1 and sessional 2 classes are derived from bio-data class. Find result in defined from sessional 1 and sessional 2 classes.

(R.G.P.V., Dec. 2010)

Ans.

```
#include <iostream.h>
const int MAX_LEN = 25
class bio-data
{
    protected :
        int roll-no;
        char branch[20];
```

Msinverter

```
    public:
    void Read_bio_data( )
    {
        cout << "Roll Number:";
        cin >> roll-no;
        cout<<"Branch Studying:";
        cin >> branch;
    }
    void Display_bio_data( )
    {
        cout <<"Roll Number:" <<roll-no<<endl;
        cout <<"Branch: "<< branch << endl;
    }
};
    class sessional1: virtual public bio-data
    {
        protected:
            int marks1, marks2;
        public:
            void ReadData( )
            {
            cout <<"Marks scored in subject1:";
            cin >> marks1;
            cout << "Marks scored in subject2:";
            cin>>marks2;
            }
            void DisplayData( )
            {
            cout <<" Sessional 1 marks scored in
                subject1 : "<<marks1 << endl;
            cout <<" Sessional1 marks scored in
                subject2 : "<<marks2 << endl;
            cout <<" Sessional total marks scored" <<
                sessional1Total( ) << endl;
            }
            int Sessional1Total( )
            {
                return marks1 + marks2;
            }
    };
    class sessional2: virtual public bio-data
    {
        protected:
            int marks1, marks2;
```



```
        public:
            void ReadData( )
            {
            cout <<"Marks scored in subject1:";
            cin >> marks1;
            cout << "Marks scored in subject2:";
            cin >> marks2;
            }
            void DisplayData( )
            {
            cout <<"sessional2 marks scored in subject1:"
                <<marks1<<endl;
            cout <<"sessional2 marks scored in
                subject2 :" << marks 2 << endl;
            cout << "sessional 2 total marks scored:"<<
                sessional2Total( ) << endl;
            }
            int sessional2Total( )
            {
                return marks1 + marks2;
            }
    };
class FinalResult: public sessional1, public sessional 2
{
    private:
        int total;
    public:
        int TotalMarks( )
        {
        total=sessional1Total+ sessional2 Total;
        return total;
        }
};
int main( )
{
    FinalResult student1;
    cout <<"Enter data for student1....." <<endl;
    student1.Read_bio_data( );
    cout << "Enter sessional1 marks....." <<endl;
    student1.sessional1 : : ReadData( );
    cout <<"Enter sessional2 marks....." << endl;
    student1.sessional2 : : ReadData( );
    cout << "Student details:";
    student1.Display_bio-data( );
    student1.sessional1 : : DisplayData( );
    student1.sessional2 : : DisplayData( );
    cout << "Total Marks =" <<student1.TotalMarks( );
}
```

MsInventer

Q.35. List the relationship used in class diagram. **(R.G.P.V., Dec. 201**

Ans. The following relationship used in class diagram –

(i) Attribute (ii) Association (iii) Navigability
(iv) Aggregation (v) Composition (vi) Generalization.

Q.36. What are the different kinds of relationship among classes Explain. **(R.G.P.V., June 201**

Or

What are the different kinds of relationships between classes ? Discu each relationship with an example. **(R.G.P.V., June 201**

Or

Explain the relationship among classes by giving suitable example. **(R.G.P.V., Dec. 2009, 201**

Or

Define different types of relationships between classes. **(R.G.P.V., Dec. 201**

Ans. Class relationships describe how classes within a program intera with each other. They are important in object-oriented design because th help to understand how a program is organized, and this understanding hel us write the actual code for the program.

When we use natural languages like English to describe object-oriente programs, we generally use verbs to describe relationships between classe while nouns are used to describe the classes themselves. The different kin of relationship among classes are as follows –

(i) Attribute – An attribute is said to model the ***has*** a relationshi We say that an employee has a name, for example.

Because the attribute is entirely enclosed within the class, we do n usually consider it when working with class diagrams.

(ii) Association – An association is the most commonly-use relationship between objects. In an association one object uses another obje to help it carry out a task. For this reason, association is often called a use relationship. Terms like "uses", "sends to", "gets from", "depends o "request", or "tells" probably indicate associations.

In UML class diagrams, associations are indicated by lines, as shown fig. 3.14.

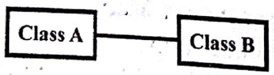


Fig. 3.14 Association

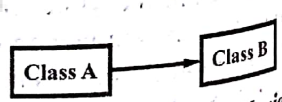


Fig. 3.15 Association with Navigabili

(iii) Navigability – Associations can go both ways at once, with bo classes being able to make requests of the other. More often they go only one w with class A asking B to help it out, but class B not asking class A for anything. T direction of an association is called navigability, and is represented on UML cl diagrams by a small open arrowhead pointing toward the collaborator (B). If each class asks the other to do something, then no arrows are used; just an unadorned line. Fig. 3.15 shows association with navigability from A to B.

(iv) Aggregation – The second major class relationship is aggregation. Aggregation means that one object (say A) contains other objects (say B, C and D). Another way to say this is that B, C, and D are part of A. Examples are an Address Book object containing many people objects, and a seminar containing many students. Aggregation is sometimes called the *part-of* relationship.

In UML terminology the object that contains the *parts* is called the *whole*. In the paragraph above, A is the whole, while B, C, and D are the parts.

In UML class diagrams, aggregation is depicted by lines from the parts meeting at an empty diamond-shaped arrowhead pointing to the whole. This is shown in fig. 3.16.

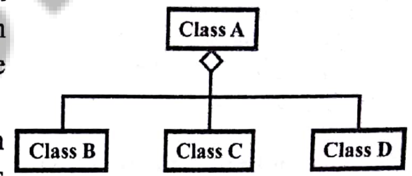


Fig. 3.16 Aggregation

(v) Composition – Composition is a stronger kind of aggregation, in which the parts are necessary to the whole and are more permanently bound to it.

Formally, a composition relationship must satisfy three criteria. First, as in aggregation, classes B, C, and D are part of A. Second B, C and D cannot be part of any other class. Third, B, C and D have lifetimes contemporaneous with A.

In UML class diagrams, composition is depicted with lines from the parts that meet of a filled-in diamond shaped arrowhead pointing to the whole. This is shown in fig. 3.17.

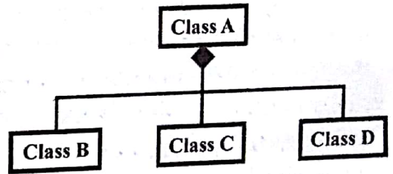


Fig. 3.17 Composition

(vi) Generalization – Generalization is another name for inheritance. In OOD the terms superclass and subclass are often used instead of base class and derived class. The superclass is a generalization of the subclass, in the same way the Mammal class is a generalization of the Tiger class. Generalization is often called the is a kind of or the is a relationship. Generalization is a relationship defined at the class level, not the object level. In UML class diagrams, it is indicated by a line from the subclass terminating in an open triangle-shaped arrowhead pointing to the superclass. Fig. 3.18 shows how this looks where B is a subclass of A.

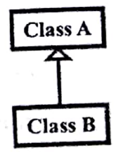


Fig. 3.18 Generalization

Q.37. What is containership ? How does it differ from inheritance ?
(R.G.P.V., June 2010, Dec. 2010)

Or

What is the diference between containership and inheritance ?
(R.G.P.V., Dec. 2011)

Ans. There is a type of relationship, called "has_a" relationship, or ***containership***. We say that starling has a tail, meaning that each bird starling includes an instance of a tail. In OOP, the "has_a" relationship occurs when one object is contained in another.

Example –

```
class alpha { ........ };
class beta {........ };
class gamma
{
    alpha a; // a is an object of alpha class
    beta b; // b is an object of beta class
    ............
};
```

All objects of gamma class will contain the objects a and b. This kind of relationship is called *containership* or *nesting*.

Whereas, in inheritance, if a class B is derived from a class A, we can say that "B is a kind of A". This is because B has all the characteristics of A and in addition some of its own. Its like saying that a starling is a kind of bird; A starling has the characteristics shared by all birds (wings, feather and so on) but has some distinctive characteristics of its own (such as dark indescent plumage). For this reason, inheritance is often called a "kind of" relationship or *derivation*.

ASSOCIATION, AGGREGATION

Q.38. What is meant by association of objects ? **(R.G.P.V., June 2014)**

Ans. Refer to Q.36 (ii).

Q.39. Explain about association between objects by giving proper examples.
(R.G.P.V., Dec. 2005, 2013)

Or

Explain the term association class with example. **(R.G.P.V., June 2012)**

Or

Explain the association between objects by giving suitable example.
(R.G.P.V., Dec. 2009, 2014)

Ans. Any dependency between two or more classes is an association. A reference from one class to another is an association. As we know, attributes should not refer to classes, use an association instead. For example, class person should not have an attribute employer, relate class person and class company by association works for, associations show dependencies between classes at the same level of abstraction as the classes themselves, while object valued attributes hide dependencies and obscure their two way nature. Associations can be implemented in various ways, but such implementation decisions should be kept out of the analysis model to preserve design freedom.

An association describes a group of links with common structure and common semantics. For example, a person ***works-for*** a company. All the links in an association connect objects from the same classes. Associations and links often appear as verbs in a problem statement. An association describes a set of potential links in the same way that a class describes a set of potential objects.

Associations are often implemented in programming languages as pointers from one object to another. A ***pointer*** is an attribute in one object that contains an explicit reference to another object. For example, a data structure for ***person*** might contain an attribute ***employer*** that points to a ***company*** object, and a ***company*** object might contain an attribute ***employees*** that points to a set of ***employee*** objects. Implementing associations as pointer is perfectly acceptable, but associations should not be modeled this way.

A link shows a relationship between two (or more) objects. Modeling a link as a pointer disguises the fact that the link is not part of either object by itself, but depends on both of them together. A company is not part of a person, and a person is not part of a company. Furthermore, using a pair of matched pointers, such as the pointer from ***person*** to ***company*** and the pointer from ***company*** to a set of ***employee,*** hides the fact that the forward and inverse pointers are dependent on each other. All connections among classes should therefore be modeled as associations, even in designs for programs. We must stress that associations are not just database constructs, although relational databases are built on the concept of associations.

Fig. 3.19 shows a one-to-one association and corresponding links. Each association in the class diagram corresponds to a set of links in the instance diagram, just as each class corresponds to a set of objects. ***Has-capital*** is the name of the association. The OMT notation for an association is a line between classes. A link is drawn as a line between objects.

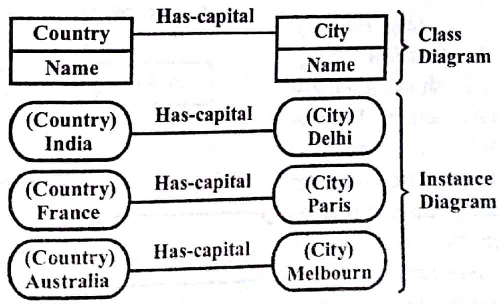


Fig. 3.19 One-to-one Association and Links

Q.40. Describe types of association. ***(R.G.P.V., Dec. 2011)***

Ans. The types of association are as follows –

(i) One-to-one Association – Fig. 3.19 shows a one-to-one association and corresponding links.

One-to-one denotes a very narrow association. Each association in a class diagram corresponds to a set of links in the instance diagram, just as each class corresponds to a set of objects. Each country has a capital city. Has capital is the name of the association. The OMT notation for an association is a line between class. A link is drawn as a line between objects. Association names are italicized. An association name may be omitted if a pair of classes has a single association whose meaning is obvious. It is good to arrange the classes to read from left-to-right, if possible.

(ii) Many-to-many Association – Fig. 3.20 is a fragment of an object model for a program, it shows many-to-many association.

In the class diagram, each point denotes the intersection of two or more lines, each line has zero or more intersection points. The instance diagram shows one possible set of lines. Lines L1, L2 and L3 intersect at point P1. Lines L3 and L4 intersect at point P2. Line L5 has no intersection points and, thus, has no link. The solid balls and "2+" are multiplicity symbols. Multiplicity specifies how many instances of one class may relate to each instance of another class.

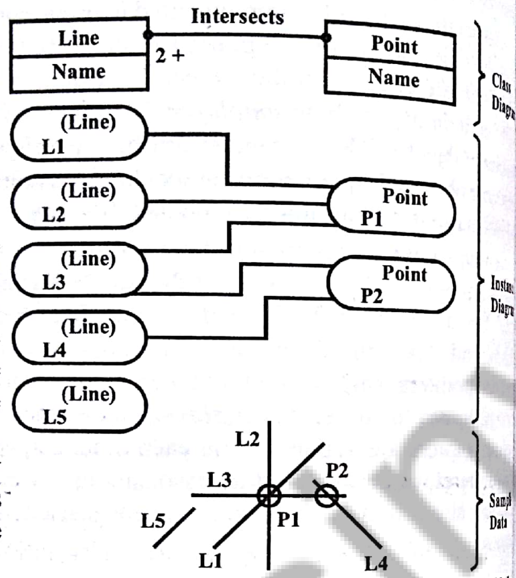


Fig. 3.20 Many-to-many Association and Links

(iii) Ternary Association – Fig. 3.21 shows a ternary association. Persons who are programmers use computer languages on projects. This ternary association is an atomic unit and cannot be subdivided into binary associations without losing information. A programmer may know a

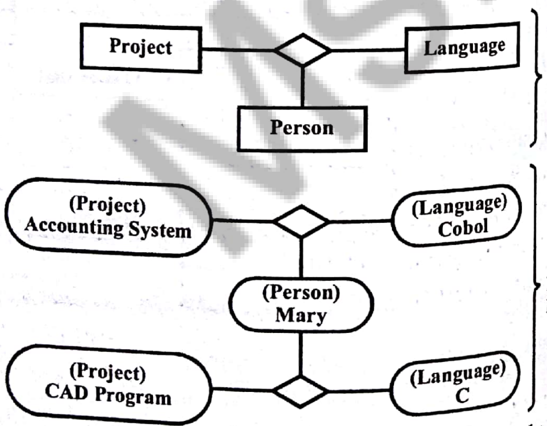


Fig. 3.21 Ternary Association and Links

language and work on a project, but might not use the language on the project. The OMT symbol for general ternary and n-array association is a diamond with lines connecting to related classes. The name of the association is written next to the diamond. Note that we did not name the association in fig. 3.21, association names are optional and a matter of modeling judgement. Associations are often left unnamed when they can be easily identified by their classes.

Q.41. What is association ? How will you identify association ? What are the different forms of association ? ***(R.G.P.V., Dec. 2010)***

Ans. **Association** – Refer to Q.36 (ii).

Identifying Associations – An association occurs where classes depend on or refer to, one another. Associations are important because they define the routes for message passing between objects, and therefore relate to the operations which objects must have to meet their responsibilities. We can derive a candidate list of associations by identifying verbs and phrases in the problem statement, although this is often less clear cut than identifying nouns.

Forms of Association – Refer to Q.40.

Q.42. Write short note on – class diagrams of one-to-one and many-to-one association with suitable example. ***(R.G.P.V., June 2002)***

Ans. **Class-diagram of One-to-one Association** – Refer to Q.40 (i).

Class-diagram of Many-to-one Association – Fig. 3.22 presents class-diagram for many-to-one association. Each person working for a company receives a salary and has a job-title. Many-to-one association is a wide association. Each association in the class diagram corresponds to a set of links in the instance diagram, just as each class corresponds to a set of objects. Many persons work for a company. Thus, works-for is the name of a many-to-one association. It has link attributes salary and job-title. The OMT notation for an association is a line between classes. The solid ball at the left end of link indicates multiplicity of association i.e., how many instances of person class are related to each instance of a company class. Here the multiplicity is many-to-one. For many-to-one association read the given fig. 3.22 from left to right.

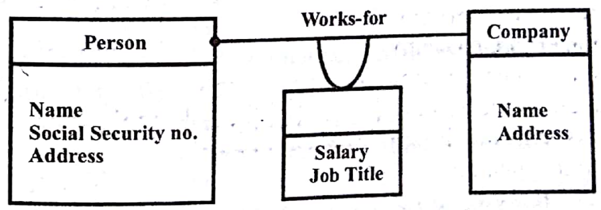


Fig. 3.22 Many-to-one Association

Q.43. What is association ? Give syntax structure of many-to-many association. ***(R.G.P.V., June 2015)***

Ans. Refer to Q.36 (ii) and Q.40 (ii).

Q.44. What is the importance of associations ?

Ans. The notion of an association is not a new concept. Associations have been widely used throughout the database modeling community for years. On the other hand, few programming languages explicitly support associations. We nevertheless emphasize that associations are a useful modeling construct for programs as well as databases and real-world systems, regardless of how they are implemented. You should not bury pointers or other object references inside objects as attributes. Instead you should model them as associations to indicate that the information they contain is not subordinate to a single class but depends on two or more classes.

Q.45. Write short note on qualified association. ***(R.G.P.V., Dec. 200)***

Ans. A qualified association relates two object classes and a qualifier. The qualifier is a special attribute that reduces the effective multiplicity of an association. One-to-many and many-to-many associations may be qualified. The qualifier distinguishes among the set of objects at the many end of an association. A qualified association can also be considered a form of ternary association.

For example, in fig. 3.23, a directory has many files. Directory and file are object classes and file name is the qualifier. A directory plus a file name yield a file. Qualification reduces the effective multiplicity of this association from one-to-many to one-to-one.

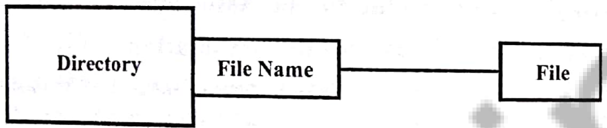


Fig. 3.23 A Qualified Association

Qualification improves semantic accuracy and increases the visibility of navigation paths. A qualifier is drawn as a small box on the end of the association line near the class it qualifies.

Q.46. Write short note on recursive association.
(R.G.P.V., Dec. 2004, June 2006, Dec. 201)
Or
Explain recursive association by giving suitable example.
(R.G.P.V., June 2007, 2010, 201)
Or
Discuss in brief of the recursive association. ***(R.G.P.V., Dec. 200)***
Or
What do you understand by recursive association ? ***(R.G.P.V., Dec. 201)***
Or
Explain recursive association. ***(R.G.P.V., June 201)***

Ans. Recursive aggregation is referred to as recursive association because aggregation is a special kind of association. A recursive association has directly or indirectly, an instance of the same kind of an association; there is unlimited number of potential levels. Fig. 3.24 depicts the example of a computer program. An aggregation of blocks is a computer program with optionally recursive compound statements; the recursion finishes with simple statement. Blocks are nested to arbitrary depth.

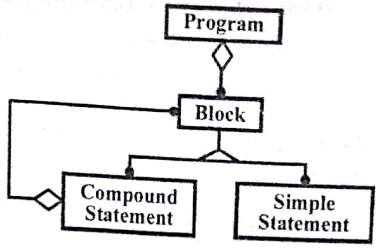


Fig. 3.24 Recursive Aggregate

The usual form of a recursive aggregate is shown in fig. 3.24 – a superclass and two subclasses, one of which is an intermediate node of the aggregate and one of which is a terminal node of the aggregate. An assembly of instances of the abstract superclass is the intermediate node.

Q.47. What are the different types of association ? Describe recursive association. ***(R.G.P.V., Dec. 2012)***

Ans. **Types of Association** – Refer to Q.40.

Recursive Association – Refer to Q.46.

Q.48. What is an association end ? What are the properties of an association end ? ***(R.G.P.V., Dec. 2016)***

Ans. End of an associaton is known as association end. A binary association has two ends and a ternary association has three ends.

Properties – Properties of an association end are given as follows –

(i) Association End Name – An association end includes name which helps to prevent multiple references to a class and provides navigation. Meaningful names create and is useful to keep the name in appropriate context.

(ii) Multiplicity – Multiplicity can be defined for each association end. "1" (exactly one), "0..1" (at most one), and "*" ("many" – zero or more) are multiplicities.

(iii) Ordering – Usually the objects on the "many" side of an association have no explicit order, and can be regarded as a set. Sometimes however, the objects are explicitly ordered.

(iv) Qualification – One or more qualifier attributes can disambiguate the objects for a "many" association end.

(v) Visibility – Association ends can be public, protected, package or private, similar to attributes and operations.

(vi) Navigability – An association can be traversed in either direction while an implementation can support one direction. The UML uses arrowhead on the association end attached to the target class to represents navigability. Arrowheads can be connected to zero, one or both ends of an association.

Msinvent

Q.49. Discuss in brief the following –
(i) Association ***(ii) Recursive association***
(iii) Many to many association ***(iv) Argument passing.***
(R.G.P.V., June 2011)

Ans. (i) Association – Refer to Q.36 (ii).

(ii) Recursive Association – Refer to Q.46.

(iii) Many to Many Association – Refer to Q.40 (ii).

(iv) Argument Passing – Parameter passing is a mechanism for communication of data and information between the calling function (caller) and the called function (callee). It can be achieved either by passing the value or address of the variable. C++ supports the following three types of parameter passing schemes –

(a) pass by value,
(b) pass by address,
(c) pass by reference (only in C++).

The parameters used to transfer data to a function are known as *input parameters* and those used to transfer the result to the caller are known as *output parameters*. The parameters used to transfer data in both the directions are called input output parameters.

Parameters can be classified as formal parameters and actual parameters. The actual parameters are those specified in the function call. The formal parameters are those specified in the function declaration and function definition. The following conditions must be satisfied for a function call –

(a) The number of arguments in the function call and the function declaration must be the same.

(b) The data type of each of the arguments in the function call should be the same as the corresponding parameter in the function declaration statement. However, the names of the argument in the function call and the parameters in the function definition can be different.

Q.50. Explain named association with example.(R.G.P.V., June 2014)

Or

Explain named association by giving suitable example.
(R.G.P.V., June 2010, 2013)

Or

Write short note on named association. (R.G.P.V., Dec. 2011, June 2015)

Ans. It is possible to give a name to an association, using a label in the centre of the line, but role names are usually preferrable. An association can have a name, we use that name to describe the nature of the relationship. Therefore, the meaning is more clear. As shown in fig. 3.25, the class person and the class company are associated, in which the name of the association is works for, that means the person class works for the company class.

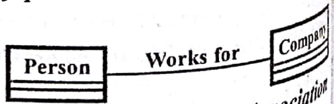


Fig. 3.25 Name of Association

Q.51. What do you mean by multiplicity ? (R.G.P.V., Dec. 2011, 2015)

Ans. Multiplicity specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity constraints the number of related objects.

Sometimes exactly one object of class A relates to exactly one object of class B. In other situations many objects of class A will relate to one object of class B. The number of objects involved on both sides of a relationship is called the ***multiplicity*** of the relationship. In class diagrams, symbols are used at both ends of the relationship line to indicate multiplicity. Table 3.1 shows the UML multiplicity symbols.

Table 3.1 The UML Multiplicity Symbols

Symbols	***Meaning***
0	none
1	one
*	some(0 to infinity)
0....1	none or one
1....*	one or more
2....4	two, three, or four
7, 11	seven or eleven

Fig. 3.26 shows some relationships between classes, with their associated multiplicities.

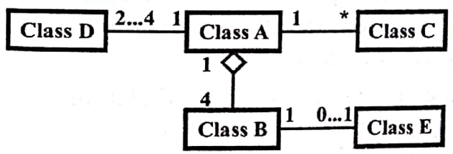


Fig. 3.26 Multiplicity

Each object of class A is an aggregate of exactly four objects of class B. Each object of class A may be associated with zero or more objects of class C and each object of class C is associated with exactly one object of class A. Each object of class A is also associated with two, three, or four objects of class D. Each object of class B may or may not be associated with one object of class E.

Q.52. Explain the multiplicity and navigability with suitable example. Draw the relationship between them. ***(R.G.P.V., June 2015, 2016)***

Ans. Refer to Q.51 and Q.36 (iii).

Relationship –

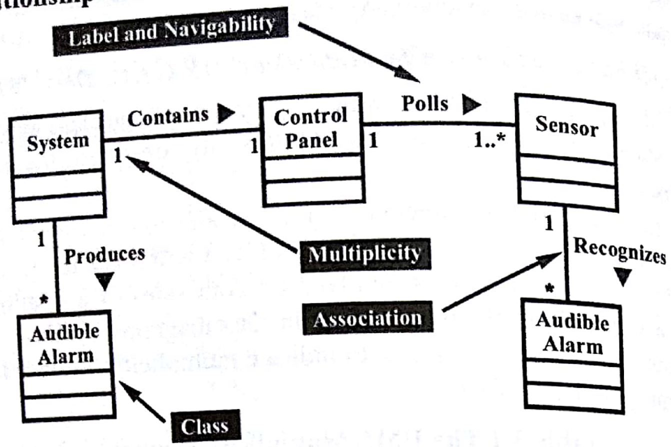


Fig. 3.27

Q.53. What are the difference between a method and message ?
(R.G.P.V., Dec. 2014)

Ans. Message communicates and method activates the execution of the message. In other words, message instructs method executes. The system displays an error message, if a method matching the message is not found in the class.

Q.54. Explain meaning of aggregate components of objects with example. ***(R.G.P.V., Dec. 2002, 2009)***

Or

Explain about aggregation. ***(R.G.P.V., June 2014)***

Ans. Aggregation is the "part-whole" or "a-part-of" relationship in which objects representing the components of something are associated with an object representing the entire assembly.

We define an aggregation relationship as relating an assembly class to one component class. Aggregation is a strong form of association in which an aggregate object is made of components. Components are part of the aggregate. The aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects. A single aggregate object may have several parts, each part-whole relationship is treated as a separate aggregation in order to emphasize the similarity to association. Parts may or may not exist apart from the aggregate or appear in multiple aggregates. Aggregation is inherently transitive, an aggregate has parts which in turn may have parts. Many aggregate operations imply transitive closure and operate on both direct and indirect parts. There are many examples of aggregate components of objects – a computer whose components are screen, keyboard, mouse and processor box, or a flower whose components are stem, leaves, head and petals.

A composition hierarchy describes an aggregation of objects which together are component parts of another object. Aggregation involves levels of abstraction, though in a different way to inheritance. Aggregation is a flexible approach to associations between objects which are not inheritance. It does not preclude the possibility of objects changing their roles in an aggregation at run time.

A 'layered model' approach to aggregation may assist in creating 'open' class libraries, portable between systems.

Aggregation may well exist in several layers, so that objects are composed of component objects which themselves are composed of other objects. Thus aggregation is also called aggregate components of objects. A train, for example, is composed of one or more locomotives and a number of coaches or wagons. The locomotives and coaches/ wagons are in turn composed of many smaller discrete components. Fig. 3.28 shows the layers of aggregation – the train as an aggregate component of discrete objects.

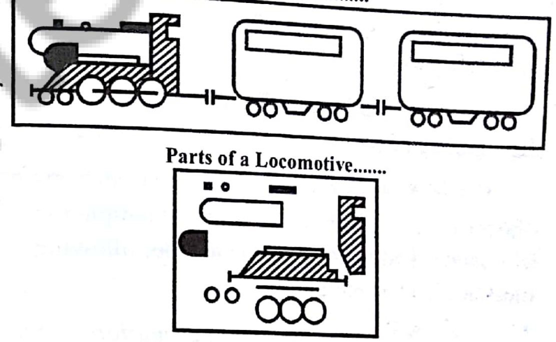


Fig. 3.28

Q.55. Write short note on modeling association and aggregation. ***(R.G.P.V., June 2013)***

Ans. Refer to Q.39 and Q.54.

Q.56. What are the properties of aggregations ?
(R.G.P.V., Dec. 2009, June 2010)

Ans. Rambaugh notes that there are certain properties associated with the objects in an aggregation –

(i) Transitivity – If A is a part of B and B is part of C, then A is part of C, e.g., if doors are parts of lounge, and lounge is part of house, then doors are parts of house.

(ii) Antisymmetry – If A is part of B, then B is not part of A, e.g., if kitchen is part of house then house is not part of kitchen.

(iii) Propagation – The environment of the part is the same as that of the assembly, e.g., if car is in garage then the steering wheel is unlikely to be somewhere else.

Q.57. Define aggregation. What are the properties of aggregation ?
(R.G.P.V., Dec. 2013)

***Ans.* Aggregation** – Refer to Q.54.

Properties of Aggregation – Refer to Q.56.

Q.58. What are the different types of aggregation ? Define them with examples. ***(R.G.P.V., June 2002, Dec. 2008, June 2012)***

Or

Explain the types of aggregation by giving suitable example. ***(R.G.P.V., Dec. 2009, 2014)***

Or

Discuss the different types of aggregation with the help of an example. ***(R.G.P.V., June 2016)***

Ans. There are three types of aggregation as follows –

(i) Fixed – The particular numbers and types of the component parts are predefined, e.g., a car has one engine, four wheels, one steering wheel etc.

(ii) Variable – The number of levels of aggregation are fixed, but the number of parts may vary, like the train.

(iii) Recursive – The object contains components of its own type, like a russian doll. A more specific example in C++ is the ability for an object to contain a pointer of its own type, allowing it to send messages to other objects of the same class.

Q.59. What is meant by aggregation ? What are the different types of aggregation ? ***(R.G.P.V., Dec. 2011)***

Or

What is aggregation ? Explain different types of aggregation with example. ***(R.G.P.V., June 2010, Dec. 2012)***

Ans. **Aggregation** – Refer to Q.54.

Types of Aggregation – Refer to Q.58.

Q.60. Write short note on aggregation vs generalization. ***(R.G.P.V., Dec. 2003, June 2004, Dec. 2015)***

Or

Discuss aggregation vs generalization. ***(R.G.P.V., Dec. 2013, June 2015)***

Ans. Aggregation is not the same thing as generalization. Aggregation relates instances. Two distinct objects are involved; one of them is a part of the other. Generalization relates classes and is a way of structuring the description of a single object. Both superclass and subclass refer to properties of a single object. With generalization, an object is simultaneously an instance of the superclass and subclass. Confusion arises because both aggregation and generalization give rise to trees through transitive closure. An aggregation tree is composed of object instances that are all part of a composite object; a generalization tree is composed of classes that describe an object. Aggregation is often called "a-part-of" relationship; generalization is often called "a-kind-of" or "is-a" relationship.

As for generalization a lamp is a fluorescent lamp or an incandescent lamp. While aggregation says that a lamp is made of base, cover, switch, wire and so on.

Q.61. Write a brief note on delegation. ***(R.G.P.V., Dec. 2012)***

Or

What do you understand by delegation ? ***(R.G.P.V., June 2016)***

Or

Define the term delegation. ***(R.G.P.V., Dec. 2016)***

Ans. Rambaugh makes the distinction between aggregation and ***delegation*** whereby the use of objects as implementation components for other objects may involve using only part of their interface. It means that an object which is used in an aggregation may not be fully utilized by the object which contains it, because only part of its behaviour might be appropriate in a particular context. This is particularly useful in situations where we might otherwise consider using inheritance, but do not want to inherit all the behaviours of a class. We may want to implement some kind of business system including account customers, and have an 'AccountCustomer' class already available. However, it might contain methods which are inappropriate for our new application because our new class is not truly 'a kind of' the existing class, but merely similar. Perhaps the existing class allows a customer to have more than one account number, and we might not want this possibility to exist in our system. If we inherit from the existing class then we will have to inherit all the attributes and methods into our new class, including behaviours which are inappropriate. Delegation gives us an alternative approach that we can delegate some of the behaviour of our new class to an object of the existing class without inheriting from it. In this way we use the containment of the object to mask any behaviours that are not needed. Our new customer class can contain an object of the 'AccountCustomer' class and use those behaviours which are appropriate to implement its own behaviours.

Q.62. Explain modeling an association as a class.

Ans. Sometimes it is useful to model an association as a class. Each link becomes one instance of the class. The link attribute box is actually a special case of an association as a class, and may have a name and operations in addition to attributes. Fig. 3.29 shows the authorization information for users on workstations. Users may be authorized on many workstations. Each authorization carries a priority and access privileges, shown as link attributes.

MSInverter

A user has a home directory for each authorized workstation, but the same home directory can be shared among several workstations or among several users. The home directory is shown as a many-to-one association between the authorization class and the directory class. It is useful to model an association as a class when links can participate in associations with other objects or when links are subject to operations.

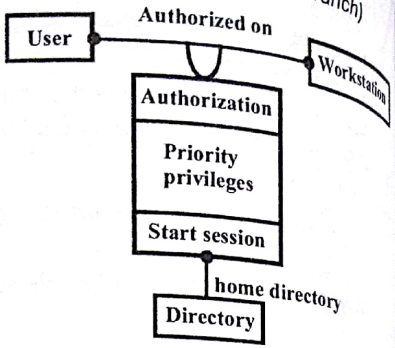


Fig. 3.29 Modeling an Association as a Class

Q.63. Compare associative aggregation and inheritance relationship.
(R.G.P.V., Dec. 2011)

Or

Write comparison between associative aggregation and inheritance relationship. ***(R.G.P.V., June 2014)***

Ans. **Aggregation** – Refer to Q.54.

Inheritance – The process by which objects of one class get the properties of object of another class is inheritance. It means that one class inherits the characteristics of another class as part of its definition. Inheritance is suitable when one class is "a kind of" other class, for example, 'text window' is a kind of 'window'. A 'classification hierarchy' shows the inheritance relationships between classes, on the basis of the sameness between different classes of objects.

Q.64. What is the difference between association, aggregation and inheritance relationships ? ***(R.G.P.V., June 2009, Dec. 2015)***

Ans. Refer to Q.39, Q.54 and Q.63.

Q.65. Compare the following –
(i) Aggregation vs generalization (ii) Aggregation vs inheritance.
(R.G.P.V., Dec. 2012)

Ans. (i) Aggregation vs Generalization – Refer to Q.60.
(ii) Aggregation vs Inheritance – Refer to Q.63.

Q.66. Explain aggregation versus association. ***(R.G.P.V., Dec. 2003)***

Or

Compare association and aggregation. ***(R.G.P.V., Dec. 2011)***

Or

Give difference between association of object and aggregation of object.
(R.G.P.V., June 2015)

Ans. Aggregation is a special form of association, not an independent concept. Aggregation adds semantic connotations in certain cases. If two objects are tightly bound by a part-whole relationship, it is an aggregation. If two objects are usually considered as independent, even though, they may often be linked, it is an association. Some tests include –

(i) Would you use the phrase *part of* ?

(ii) Are some operations on the whole automatically applied to its parts ?

(iii) Are some attribute values propagated from the whole to all or some parts.

(iv) Is there an intrinsic asymmetry to the association, where one object class is subordinate to the other ?

Aggregations includes part explosions and expansions of an object into constituent parts. In fig. 3.30, a company is an aggregation of its divisions which are in turn aggregations of their departments; a company is indirectly an aggregation of departments. A company is not an aggregation of its employees, since company and person are independent objects of equal stature.

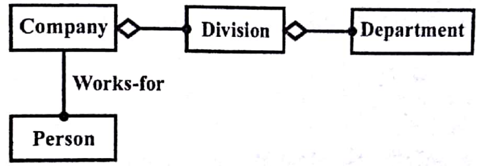


Fig. 3.30 Aggregation and Association

The decision to use aggregation is a matter of judgement and is often arbitrary. Often it is not obvious if an association should be modeled as an aggregation. To a large extent this kind of uncertainty is typical of modeling; modeling requires seasoned judgement and there are few hard and fast rules. If we exercise careful judgement and are consistent, then imprecise distinction between aggregation and ordinary association does not cause problems in practice.

Q.67. What do you understand by mandatory profile ?
(R.G.P.V., Dec. 2013)

Ans. Mandatory profiles are simply read-only versions of the standard roaming profiles.

Creating a mandatory profile is probably one of the simplest administrative tasks. If we look in each individual profiles directory, we will find a hidden file called NTUSER.DAT. This file contains all the user-configurable aspects of the profile. To create a mandatory profile, simply rename this file NTUSER.MAN.

A roaming profile follows users from computer to computer. A mandatory profile is nothing more than a roaming profile to which the user has read-only access. The only real difference between the two types of profiles is that a

user can make changes to a roaming profile, but not to a mandatory profile. Any changes a user makes while logged in with a mandatory profile are gone the next time the user logs in.

Terminal server offers significant system life cycle cost advantage including reduction of workstation software administration cost. To take full advantage of these saving, it often makes sense to use mandatory user profiles in conjunction with the thin-client. Mandatory user profiles prevent the end user from saving configuration changes to their environment. A single user profiles can be implemented for the entire organization, ensuring uniformity and reliability as well as deployment simplicity.

This system is not without drawbacks. Many applications require unique configurations for each user. In a non-mandatory environment, such configurations become costly without automation. In a mandatory profile environment, configuration automation becomes critical. Without it, each user is forced to self-configured at each logon because settings are not saved.

Messaging applications have the common characteristic of always containing configuration setting unique to each user. Profile maker is a powerful tool that provides the administrator with a flexible solution to this problem. Normally profile maker deployment consists of the installation of several files to the Netlogon share on a domain controller. The installation is scalable and can be distributed to all domain controllers using replication. The background configuration component is generally launched from a logon script. Although not an issue for terminal server, there is no workstation setup required.

In many environment terminal server is the solution of choice for reducing administration costs. It makes sense to implement mandatory user profiles for the same reason. Profile maker uses "server-based profile control" to bridge the gap between configuration requirements and end user restrictions so that we can have the best of both worlds.

CONCEPT OF INTERFACES AND ABSTRACT CLASSES

Q.68. Define interfaces in Java. How interfaces are implemented ? How can they be accessed ? How to apply interfaces ? What is meant by extension of interfaces ? Explain with an example. ***(R.G.P.V., June 2010, Dec. 2010)***

Or

Explain the interface concept in Java. ***(R.G.P.V., June 2010)***

Or

Explain the interface concept in Java with example. ***(R.G.P.V., June 2010)***

Or

Define interfaces in Java. ***(R.G.P.V., June 2010)***

Ans. **Defining an Interface** – An interface is basically a type of class. Defining an interface is just like defining a class. The general form of an interface is –

```
access interface interfacename
{
    variables declaration;
    methods declaration;
}
```

When access specifier is not, then default access results, and the interface is only available to other members of the package in which it is declared. When access specifier is public, the interface is used by any other code. In this case, the interface must be the only public interface declared in the file, and the file name and the interface. Interfacename is the name of the interface and is any valid identifier. It is noted that declared methods do not have bodies. They terminate with a semicolon after the parameter list. They are abstract methods. Any method specified within an interface has no default implementation. Each class that has an interface must implement all of the methods. Variables are declared inside of interface declarations. They are implicitly final and static. It means that they cannot be altered by the implementing class. They must also be initialized. All variables and methods are implicitly public. An example of an interface definition is as follows –

```
interface Callback
{
    void callback(int param);
}
```

Implementation – After defining an interface, one or more classes can implement the interface. Include the ***implements*** clause in a class definition and then create the methods defined by the interface to implement an interface. The general form of implementation may look like as follows –

```
class classname[extends superclass]
[implements interface[,interface....]]
{
    //class_body
}
```

The interfaces are separated with a comma, if a class implements more than one interface. When a class implements two interfaces that declare the identical method, then the same method will be used by clients of either interface. The methods that implement an interface are declared as ***public***. Also, the type signature of the implementing method is the same as the type signature specified in the interface definition.

An example class that implements the Callback interface is shown below.

```
class Client implements Callback
{
    public void callback(int p)
    {
    System.out.println("callback called with" + p);
    }
}
```

Accessing Implementations Through Interface References – Variables are declared as object references that use an interface instead of a class type. Any instance of any class that implements the declared interface is referred to by such a variable. If a method is called using one of these references, the correct version will be called on the basis of the actual instance of the interface being referred to. This is an important feature of interfaces. The method to be executed is looked up dynamically at run time. This permits classes to be created later than the code which calls methods on them. Without having to know anything about the "Callee" the calling code can dispatch through an interface. Here is an example that calls the Callback( ) method through an interface reference variable –

```
Class TestIface
{
    public static void main(String args[ ])
    {
        Callback c = new Client( )
        c.Callback(32);
    }
}
```

Applying Interfaces – There are a number ways to implement a stack. For instance, the stack can be growable or it can be of a fixed size. The stack is also held in an array, a linked list, a binary tree, etc. Consider, for example, an interface that defines an integer stack. Keep this in a file called IntStack.Java. This interface is used by both stack implementations.

```
//Define an integer stack interface
interface IntegerStack
{
    void push(int item);   //store an item
    int pop( );            //retrieve an item
}
```


The following program creates a class called ***FixedStack*** which implements a fixed-length version of an integer stack –

```
//An implementation of IntStack that uses fixed storage
class FixedStack implements IntegerStack {
    private int stack [ ];
    private int tos;
//allocate and initialize stack
FixedStack (int MAX) {
    stack = new int [MAX];
    tos = –1;
}
//Push an item onto the stack
public void push (int item) {
    if (tos = = stack.length–1) //use length member
        System.out.println ("Stack is full.");
    else
        stack [++ tos] = item;
    }
    //Pop  an item from the stack
    public int pop( ) {
        if (tos < 0) {
            System.out.println("Stack empty.");
            return 0;
        }
        else
            return stack [tos– –];
    }
}
class IFTest {
    public static void main (string args [ ]) {
        FixedStack stack1 = new FixedStack (6);
        FixedStack stack2 = new FixedStack (10);
        //push some numbers onto the stack
        for (int i = 0; i < 6; i++) stack1.push(i);
        for (int i = 0; i < 10; i++) stack2.push(i);
        //pop those numbers off the stack
            System.out.println("Stack in stack1:");
        for(int i = 0; i < 6; i++)
            System.out.println(stack1.pop( ));
            System.out.println("Stack in stack2:");
        for(int i = 0; i < 8; i++)
            System.out.println(stack2.pop( ));
    }
}
```

Extending Interfaces – Interfaces can also be extended. It means that an interface is subinterfaced from other interfaces. The new subinterface inherits all the members of the superinterface in the way identical to subclasses. This is done using the keyword *extends* as shown below –

```
interface name2 extends name1
{
    body of name2
}
```

For instance, we can put all the methods in one interface and the constants in the other. This will enable us to use the constants in classes where the methods are not needed. For example,

```
interface ItemConstants
{
    int code = 500;
    String name = "cooler";
}
interface Item extends ItemConstants
{
    void display( );
}
```

Q.69. Describe the various forms of implementing interfaces. Give examples of Java code for each case. ***(R.G.P.V., June 2011)***

Ans. Refer to Q.68.

Q.70. How many interface can a class inherit and can a same interface be inherited by two different classes ? Explain it.

Ans. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces that do not make assumptions about how they are implemented. Once it is defined, any number of classes can implement an *interface*. Also, one class can implement any number of interfaces. By providing the *interface* keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface.

Q.71. What does Java API consists of ?

Or

What are the basic functions of Java Application Programming Interface (API) ?

Ans. Java API is a list of all classes that are part of the Java development kit. It includes all packages, classes and interfaces along with their methods, fields and constructors. These prewritten classes provide number of functionality to a programmer. In Java most basic programming tasks are performed by the API's classes and packages, which are helpful in minimizing the number of lines written within pieces of code. Most commonly used packages are as follows –

(i) ***Language Support Package*** – A collection of classes and methods required for implementing basic features of Java.

(ii) ***Utilities Package*** – A collection of classes to provide utility functions such as date and time functions.

(iii) ***Input/Output Packages*** – A collection of classes required for input/output manipulation.

(iv) ***Networking Package*** – A collection of classes for communicating with other computers via internet.

(v) ***AWT Package*** – The abstract window tool kit package contains classes that implements platform-independent graphical user interface.

(vi) ***Applet Package*** – This include a set of classes that allows us to create Java applets.

Q.72. Define variable in interfaces.

Ans. Interfaces use the import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class, all of those variable names will be in scope as constants. If an interface contains no methods, then any class that includes such an interface does not actually implement anything. It is as if that class were importing the constant fields into the class name space as final variables. Here example uses this technique to implement an automated "decision maker" –

```
import java.util.Random;
interface SharedConstants {
    int NO = 0;
    int YES = 1;
    int MAYBE = 2;
    int LATER = 3;
    int SOON = 4;
    int NEVER = 5;
}
```

```
class Question implements SharedConstants {
    Random rand = new Random( );
    int ask( ) {
        int prob = (int) (100 * rand.nextDouble( ));
        if(prob < 30)
            return NO;              // 30%
        else if(prob < 60)
            return YES;             // 30%
        else if(prob < 75)
            return LATER;           // 15%
        else if(prob < 98)
            return SOON;            // 13%
        else
            return NEVER;           // 2%
    }
}
class AskMe implements SharedConstants {
    static void answer(int result) {
        switch(result) {
            case NO;
                System.out.println("No");
                break;
            case YES;
                System.out.println("Yes");
                break;
            case MAYBE;
                System.out.println("Maybe");
                break;
            case LATER;
                System.out.println("Later");
                break;
            case SOON;
                System.out.println("Soon");
                break;
            case NEVER;
                System.out.println("Never");
                break;
        }
    }
    public static void main(String args[ ]) {
        Question q = new Question( );
        answer(q.ask( ));
        answer(q.ask( ));
        answer(q.ask( ));
        answer(q.ask( ));
    }
}
```


In this program makes use of one of Java's standard classes, *Random*. This class provides pseudorandom numbers. It contains several methods that allow you to obtain random numbers in the form required by your program.

Here, the two classes, *Question* and *AskMe*, both implement the *SharedConstants* interface where *NO, YES, MAYBE, SOON, LATER*, and *NEVER* are defined. Here is the output of a sample run of this program. Note that the results are different each time it is run.

```
Later
Soon
No
Yes
```

Q.73. Discuss the relationship between event classes and interfaces with an appropriate diagram and describe types of listener interfaces.

Ans. Within the limits of the events that the AWT knows about, you completely control how events are transmitted from the event sources, such as buttons or scrollbars, to event listeners. You can designate any object to be an event listener – in practice you pick an object that can conveniently carry out the desired response to the event.

Event sources have methods that allow you to register event listeners with them. When an event happens to the source, the source sends a notification of that event to all the listener objects that were registered for that event.

In an object-oriented language like Java, the information about the event is encapsulated in an event object. In Java, all event objects ultimately derive from the class java.util.Event Object. Of course, there are subclasses for each event type, such as *ActionEvent* and *WindowEvent*. Different event sources can produce different kinds of events. For example, a button can send *ActionEvent* objects, whereas a window can send *WindowEvent* objects.

To sum up, here's an overview of how event handling in the AWT works –

(i) A listener object is an instance of a class that implements a special interface called a *Listener interface*.

(ii) An event source is an object that can register listener objects and send them event objects.

(iii) The event source sends out event objects to all registered listeners when that event occurs.

(iv) The listener objects will use the information in the event object to determine their reaction to the event.

The delegation event model has two parts – *sources* and *listeners*. Listeners are created by implementing one or more of the interfaces defined by the *java.awt.event* package. When an event occurs, the event source invokes the appropriate method defined by the listener and provides an event object as its

argument. Table 3.2 lists commonly used listener interfaces and provides a brief description of the methods that they define.

Table 3.2 Commonly used Event Listener Interfaces

Interface	*Description*
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

The specific methods that are contained in each interface are discussed below –

(i) ActionListener Interface – This interface defines the *actionPerformed( )* method that is invoked when an action event occurs. Its general form is shown below –

void actionPerformed (ActionEvent *ae*)

(ii) AdjustmentListener Interface – This interface defines the *adjustmentValueChanged( )* method that is invoked when an adjustment event occurs. Its general form is shown below –

void adjustmentValueChanged (AdjustmentEvent *ae*)

(iii) ComponentListener Interface – The interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown below –

void componentResized(ComponentEvent *ce*)
void componentMoved(ComponentEvent *ce*)
void componentShown(ComponentEvent *ce*)
void componentHidden(ComponentEvent *ce*)

(iv) ContainerListener Interface – This interface contains two methods. When a component is added to a container, *componentAdded( )* is invoked. When a component is removed from a container, *componentRemoved( )* is invoked. Their general forms are shown below –

void componentAdded(ContainerEvent *ce*)
void componentRemoved(ContainerEvent *ce*)

(v) FocusListener Interface – This interface defines two methods. When a component obtains keyboard focus, *focusGained( )* is invoked. When a component loses keyboard focus, *focusLost( )* is called. Their general forms are shown below –

void focusGained(FocusEvent *fe*)
void focusLost(FocusEvent *fe*)

(vi) ItemListener Interface – This interface defines the *itemStateChanged( )* method that is invoked when the state of an item changes. Its general form is shown below –

void itemStateChanged(ItemEvent *ie*)

(vii) KeyListener Interface – This interface defines three methods. The *keyPressed( )* and *keyReleased( )* methods are invoked when a key is pressed and released, respectively. The *keyTyped( )* method is invoked when a character has been entered. Their general forms are shown below –

void keyPressed(KeyEvent *ke*)
void keyReleased(KeyEvent *ke*)
void keyTyped(KeyEvent *ke*)

(viii) MouseListener Interface – This interface defines five methods. If the mouse is pressed and released at the same point, *mouseClicked( )* is invoked. When the mouse enters a component, the *mouseEntered( )* method is called. When it leaves, *mouseExited( )* is called. The *mousePressed( )* and *mouseReleased( )* methods are invoked when the mouse is pressed and released, respectively. Their general forms are shown below –

void mouseClicked(MouseEvent *me*)
void mouseEntered(MouseEvent *me*)
void mouseExited(MouseEvent *me*)
void mousePressed(MouseEvent *me*)
void mouseReleased(MouseEvent *me*)

(ix) MouseMotionListener Interface – This interface defines two methods. The *mouseDragged( )* method is called multiple times as the mouse

is dragged. The *mouseMoved( )* method is called multiple times as the mouse is moved. Their general forms are shown below –

void mouseDragged(MouseEvent *me*)
void mouseMoved(MouseEvent *me*)

(x) MouseWheelListener Interface – This interface defines the *mouseWheelMoved( )* method that is invoked when the mouseWheel is moved. Its general form is shown below –

void mouseWheelMoved(MouseWheelEvent *mwe*)

(xi) TextListener Interface – This interface defines the *textChanged* method that is invoked when a change occurs in a text field or text area. Its general form is shown below –

void textChanged(TextEvent *te*)

(xii) WindowFocusListener Interface – This interface defines two methods – *windowGainedFocus( )* and *windowLostFocus( )*. These are called when a window gains or loses input focus. Their general forms are shown below –

void windowGainedFocus(WindowEvent *we*)
void windowLostFocus(WindowEvent *we*)

(xiii) WindowListener Interface – This interface defines seven methods. The *windowActivated( )* and *windowDeactivated( )* methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the *windowIconified( )* method is called. When a window is deiconified, the *windowDeiconified( )* method is called. When a window is opened or closed, the *windowOpened( )* or *windowClosed( )* methods are called, respectively. The *windowClosing( )* method is called when a window is being closed. Their general forms are shown below –

void windowActivated(WindowEvent *we*)
void windowClosed(WindowEvent *we*)
void windowClosing(WindowEvent *we*)
void windowDeactivated(WindowEvent *we*)
void windowDeiconified(WindowEvent *we*)
void windowIconified(WindowEvent *we*)
void windowOpened(WindowEvent *we*)

Q.74. What is abstract class and why it is needed ?

(R.G.P.V., June 201)

Or

What is meant by abstract class. ***(R.G.P.V., Nov. 201)***

Ans. A class containing at least one pure virtual is called an abstract class. Abstract classes are frequently used to define methods to be inherited by subclasses. An abstract class has no instances but whose descendent class have direct instances.

An abstract class exists for the purpose of inheriting by other classes. It does not serve any useful purpose by itself, because it is incomplete in the sense that the pure virtual function contained in it has no body. By this, we cannot create an object for the abstract class, however, we can create a pointer of this class.

Q.75. What are the properties of abstract base class ?

(R.G.P.V., Dec. 2006)

Ans. An abstract class with one or more pure virtual functions has the following properties –

(i) Describes an unrealized concept (which is yet to be conceived)
(ii) Objects of an abstract class type cannot be created
(iii) Derived classes can be built from these abstract classes
(iv) Objects of the derived classes can be created provided these derived classes do not have any pure virtual functions.

Q.76. What is an abstract use case ? Give example. (R.G.P.V., Dec. 2014)

Ans. Use cases could be viewed as concrete or abstract. An abstract use case is not complete and has no actors that initiate it but is used by another use case. This inheritance could be used in several levels. Abstract use cases also are the ones that have uses or extends relationships.

Q.77. Explain abstract methods and classes. (R.G.P.V., Dec. 2008, 2012)

Ans. An abstract method has no implementation. It exists so that its name can be inherited by derived classes. This creates a consistent interface across all the classes in a hierarchy. For example, the "turnOn" methods of fuel and water pumps are very different. Therefore the definitions of the two methods

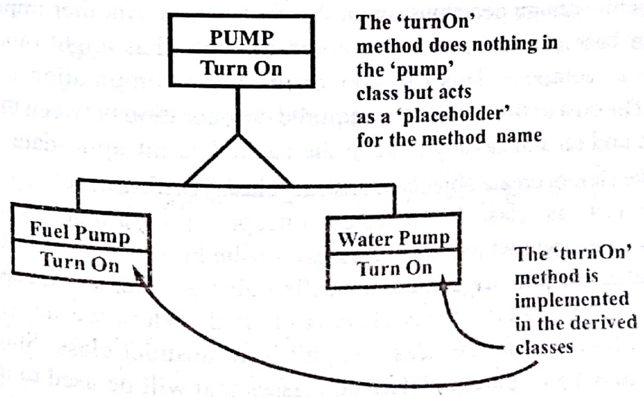


Fig. 3.31

have nothing in common. However, for reasons of design the 'Pump' class should have a 'turnOn' method. This is because it is not good practice to have overloaded method names in separate parts of a class hierarchy which do not have a common not, since the methods are then applied in an 'ad hoc' way with no reference to each other. The question, what is the implementation of the 'turnOn' method in the 'Pump' class if it is not used for common elements of the two other methods ? In fact, the method does nothing functional at all – it exists as a 'placeholder' for the name of the method so that all derived classes may implement their own versions of 'turnOn' (fig. 3.31). It is, to all intents and purposes, an 'abstract' method i.e., one which has no implementation. Such methods are an important design artifact, so just because a method does nothing, it does not mean it should not be declared. Abstract methods are common components of an abstract class.

Classes – Refer to Q.74.

Q.78. What is the importance of abstract base classes ? Can we create objects of abstract class ? Write a program to demonstrate it. ***(R.G.P.V., June 2006)***

Ans. **Importance of Abstract Base Classes** – When a class is designed only to act as a base class, to be inherited by other classes, then that class is known as abstract base class. This class is used only for deriving other classes. It is a concept for designing in the program development and it provides a base upon which other classes may be built.

An abstract base class is a base class containing pure virtual functions. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

In a very simple sense, an abstract base class serves to capture all of the relevant public design decisions about the abstraction. Another important use of abstract base classes is to cache common state that might otherwise be expensive to compute. This can convert an O(n) computation to an O(1) retrieval. The cost of this style is the required co-operation between the abstract base class and its subclasses, to keep the cached result up-to-date.

No, we cannot create objects of abstract class. An abstract class is designed only to act as a base class. It is a design concept in program development and provides a base upon which other classes may be built. Classes used only for deriving other classes are sometimes called abstract classes, means that no actual instance (object) of this class is created. When we never want to instantiate objects of a base class we call it an abstract class. Such a class exists only to act as a parent of derived classes that will be used to instantiate objects. It may also provide an interface for the class hierarchy.


```
// A program for using the abstract class
#include <iostream.h>
using namespace std;
class Student
{
        int rollno;
        public
            void getdata(int x)
            { rollno = x;
            }
            void putdata(void)
            {
                cout << "Roll Number : " << rollno << "\n";
            }
};
class Engineering : public Student
{
        float sub1, sub2;
        public :
            void getmarks(float m1, float m2)
            {
                sub1 = m1;
                sub2 = m2;
            }
        void putmarks(void)
        {
            cout << "Subject 1 = " << sub1 << "\n";
            cout << "Subject 2 = " <<sub2 << "\n";
        }
};
class Medical : public Student
{
        float sub1, sub2;
        public :
            void getmarks(float m1, float m2)
            {
                sub1 = m1;
                sub 2 = m2;
            }
            void putmarks(void)
            {
                cout << "Subject 1 = " << sub1;
                cout << "Subject 2 = " << sub2;
            }
};
```

```
int main( )
{
    Student *p;
    Engineering e;
    Medical m ;
    e.getdata(1001);
    p = &e;
    p -> putdata( );
    p -> getmarks(50.6, 60.8);
    p -> putmarks( );
    m.getdata(2001);
    p = &m;
    p -> putdata( );
    p -> getmarks(90.4, 89.7);
    p -> putmarks( );
    return 0;
}
```

Q.79. What is the difference between abstract class and interface? When should you use an abstract class, when an interface, when both? ***(R.G.P.V., June 2017)***

Ans. An abstract class contains executable methods and abstract methods. It is designed only to act as a base class. Where as an interface defines only abstract methods and fields. This means that the interface does not specify any code to implement these methods and data fields contain only constants.

An abstract class should use in case where you want to use implementation inheritance. Abstract classes are excellent candidates inside of application frameworks. Abstract classes let you define some default behaviour and force subclasses to provide any specific behaviour.

An interface is used for polymorphic interface inheritance. Where the client does not care about the actual implementation and wants to only deal with a type

UNIT 4

POLYMORPHISM – INTRODUCTION, METHOD OVERRIDING AND OVERLOADING, STATIC AND RUN TIME POLYMORPHISM

Q.1. Write short note on polymorphism. ***(R.G.P.V., Dec. 2009)***

Or

Explain the function polymorphism. ***(R.G.P.V., Nov. 2018)***

Ans. Polymorphism is an important concept of object-oriented programming. It means, the ability to take more than one form i.e., polymorphism literally means "many forms". An operation may exhibit different behaviours in different instances. These behaviours depend upon the types of data used in the operation. For example, two numbers are added and the operation will generate sum. If the operands are strings, then the operation would produce a third string by concatenation.

Polymorphism refers to the ability to perform operations without knowing the type of object that will be operated on. Polymorphism allows the system to automatically select the correct operations for the object concerned at run time. The system can dynamically choose and perform the correct operation depending on the data at run-time.

Polymorphism allows different object to respond to the same message in different ways, the response specific to type of object. This is achieved by various types of 'overloading'. It is implemented using overloaded functions and overloaded operators.

Q.2. What is inheritance and polymorphism in object oriented programming ? ***(R.G.P.V., June 2015)***

Ans. **Inheritance** – Refer to Q.1 (Unit-III).

Polymorphism – Refer to Q.1.

Q.3. What are the advantages of polymorphism ?(R.G.P.V., Dec. 2009)

Ans. The advantages of polymorphism are as follows –

(i) The biggest advantage of polymorphism is creation of reusable code by programmers, classes once written, tested and implemented can be easily reused without caring about what's written in the cases.

(ii) Polymorphic variables help with memory use, in that a single variable can be used to store multiple data types (integers, strings etc.), rather than declaring a different variable for each data format to be used.

Q.4. What are the disadvantages of polymorphism ?

(R.G.P.V., Dec. 2009)

Ans. The disadvantages of polymorphism are as follows –

(i) It's esoteric. Not very easy for the beginner to just pick up and go with it. Rather it takes often years of dedication before abstraction becomes second nature.

(ii) Huge up front hit in migrating legacy codes as it often has to be redesigned from scratch. Of course in the long run, in terms of reuse, performance and robustness, this eventually becomes an advantage.

Q.5. What are different types of polymorphism ? Describe them.

(R.G.P.V., Dec. 2008)

Or

How polymorphism is achieved at compile time and run-time ?

(R.G.P.V., June 2013, Dec. 2010)

Or

Distinguish between early and late binding. (R.G.P.V., Dec. 2010)

Or

Give the differences between static and dynamic polymorphism.

(R.G.P.V., June 2010)

Ans. There are two types of polymorphism, they are as follows –

(i) Compile-time Polymorphism – The overloaded member functions are 'selected' for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select appropriate function for a particular call at compile-time itself. This is called ***early binding*** or static binding or ***static linking***, also known as compile-time polymorphism. Early binding simply means that an object is bound to its function call at compile time.

(ii) Run-time Polymorphism – If the appropriate member function could be selected while the program is running. This is known as run-time polymorphism. C++ supports a mechanism known as virtual function to achieve run-time polymorphism.

At run-time, when class objects under consideration are known, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after compilation, this process is termed as ***late binding***. It is also known as ***dynamic binding*** because the selection of the appropriate function is done dynamically at run time.

Q.6. Explain the meaning of polymorphism. How is polymorphism achieved at run time ? Explain with coding. (R.G.P.V., June 2011, 2014)

Or

What is polymorphism ? Explain about static and dynamic polymorphism with examples. (R.G.P.V., Dec. 2011)

Or

Explain polymorphism. Differentiate between static and dynamic polymorphism with an example. (R.G.P.V., June 2012)

Or

What is polymorphism ? Compare static vs dynamic polymorphism.

(R.G.P.V., Dec. 2012)

Or

Write a program to demonstrate run-time polymorphism.

(R.G.P.V., Dec. 2012)

Or

What is polymorphism ? What are different forms of its ? Give example of each. (R.G.P.V., June 2015)

Ans. **Polymorphism** – Refer to Q.1.

Static and Dynamic Polymorphism – Refer to Q.5.

Program to Demonstrate Run-time Polymorphism –

```
//Run-time polymorphism
#include <iostream.h>
#include <cstring.h>
using namespace std;
class media
{
  protected :
        char title[55];
        float price;
  public :
        media(char*s, float a)
        {
              strcpy(title, s);
              price = a;
        }
        virtual void display( ) { }  //empty virtual function
};
```

```
class book : public media
{
    int pages;
    public:
        book(char*s, float a, int p) : media(s, a)
        {
            pages = p;
        }
        void display( );
};
class tape : public media
{
    float time;
    public:
        tape (char*s, float a, float t) : media(s, a)
        {
            time = t;
        }
        void display( );
};
void book : : display( )
{
    cout <<"\n Title :" <<title;
    cout <<"\n Pages :" <<pages;
    cout <<"\n Price :" <<price;
}
void tape : : display( )
{
    cout <<"\n Title :"<<title;
    cout <<"\n play time :" <<time <<"mins";
    cout <<"\n price :" << price;
}
int main( )
{
    char *title = new char[30];
    float price, time;
    int pages;
    //Book details
    cout <<"Enter book details\n";
    cout <<"Title :"; cin >> title;
    cout <<"Price :"; cin >> price;
    cout <<"Pages :"; cin >> pages;
    book book1(title, price, pages);
```



```
    //Tape details
    cout <<"\n Enter tape details\n";
    cout <<"Title :"; cin >> title;
    cout <<"price :"; cin >> price;
    cout <<"play time(mins):"; cin >> time;
    tape tape1(title, price, time);
    media*list[2];
    list[0] =&book1;
    list[1]=&tape1;
    cout <<"\n Media Details";
    cout <<"\n Book";
    list [0]-> display( );      //display book details
    cout <<"\n Tape";
    list[1] -> display( );      //display tape details
    result(0);
}
```

Q.7. What is run-time polymorphism ? Explain with example.
(R.G.P.V., Nov. 2018)

Ans. Refer to Q.5 and Q.6.

Q.8. Discuss about the following –
(i) Method lookup (ii) Polymorphism. ***(R.G.P.V., June 2010)***

Or

Write short note on method lookup. ***(R.G.P.V., June 2006)***

Ans. (i) Method Lookup – Method lookup is the process of determining which method definition a method signature denotes during run time, based on the type of the object. When we use the same method name in both the base and derived classes, the method in base class is declared as ***virtual*** using the keyword ***virtual*** preceding its normal declaration. When a function is made ***virtual,*** C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the ***virtual*** function. Following program illustrates this approach –

```
#include <iostream.h>
class Base
{
    public:
        void display( ) {cout<<"\n Display base";}
        virtual void show( ) {cout <<"\n Show base";}
};
class Derived : public Base
```

MsInventer

```
{
    public:
        void display( ) {cout << "\n Display derived";}
        void show( ) {cout <<"\n Show derived";}
};
int main( )
{
    Base B;
    Derived D;
    Base *bptr;
    cout<<"\n bptr points to Base\n";
    bptr = &B;
    bptr -> display( );          //calls Base version
    bptr -> show( );             // calls Base version
    cout<<"\n\n bptr points to Derived\n";
    bptr = &D;
    bptr -> display( );          //calls Base version
    bptr -> show( );             //calls Derived version
    return 0;
}
```

Here, when **bptr** is made to point to the object D, the statement

bptr → display( )

calls only the function associated with the **Base** (i.e. **Base** : : display( )) whereas the statement

bptr → show( );

calls the **Derived** version of show( ). This is because the function display( ) has not been made **virtual** in the **Base** class.

(ii) Polymorphism – Refer to Q.1.

Q.9. How can a derived function use a virtual function ? In which class should a virtual function be declared and defined.

(R.G.P.V., Dec. 2016)

Ans. Refer to Q.8 (i).

Q.10. What is polymorphism ? Give an example to illustrate that public member data of the base class becomes the private class member data of the derived class and the base class member data cannot be accessed in the main( ) function.

(R.G.P.V., Dec. 2016)

Ans. **Polymorphism** – Refer to Q.1.

Example –

```
#include <iostream.h>
class P
{
    int x;
    public:
        int y;
        void get_xy( )
        {
            cout <<"Enter values for x and y:";
            cin >> x >> y;
        }
        int get_x( )
        {
            return x;
        }
};
class Q : private P
{
    int z ;
    public :
    void mul( )
    {
        get_xy( );
        z = get_x( ) *y;
    }
    void display( )
    {
        cout << "x = "<< x << "\n";
        cout << "y = "<<y <<"\n";
        cout << "z = "<< z << "\n";
    }
};
    int main( )
    {
        Q q;
        //q.get_xy( );       would not work
        q.mul( );
        q.display( );
        return 0;
    }
```

Q.11. Explain following special feature of object oriented system with example –

(i) Encapsulation and polymorphism
(ii) Inheritance
(iii) Aggregation
(iv) Association.

(R.G.P.V., Dec. 2014)

MSInventer

Ans. (i) ***Encapsulation*** – Refer to Q.1 (Unit-II).
Polymorphism – Refer to Q.1.
(ii) ***Inheritance*** – Refer to Q.1 (Unit-III).
(iii) ***Aggregation*** – Refer to Q.54 (Unit-III).
(iv) ***Association*** – Refer to Q.36 (ii) (Unit-III).

Q.12. What do you mean by dynamic binding ? How is it useful in OOP ? ***(R.G.P.V., June 2013)***

Ans. **Dynamic Binding** – Refer to Q.65 (ii).

Dynamic binding helps to optimize the look-up mechanism in OOP to make it more efficient. As long as the class structure remains unchanged during program execution, the correct method for every operation can be stored locally in the subclass. With this technique, known as method catching, dynamic binding can be reduced to a single hash table look-up and performed in constant time regardless of the depth of the inheritance tree. Clever compiler tricks can reduce even the size of the constant cost. Thus, in OOPL, with dynamic binding the cost of run time method resolution can be reduced to a single structure reference, almost insignificant compared to the cost of calling a procedure.

Q.13. Explain virtual function.(R.G.P.V., June 2005, Dec. 2008, 2014)

Or

What is virtual function, define with example ? (R.G.P.V., Dec. 2015)

Ans. Virtual means existing in appearance but not in reality. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class.

When we use the same function name in both the base and derived classes, the function in base class is declared as virtual using the keyword virtual preceding its normal declaration. Virtual functions are the C++ way to obtain polymorphism. They provide a method of encapsulating the implementation details of an inheritance hierarchy.

Here is an example of a virtual function declaration –

```
Class A
{
    private :
        // member variable declaration
    public :
        // member function declaration
    virtual void vir_func( )
    {
        // Body of function
    }
}
```


Q.14. What is virtual function ? Why do we need virtual function ?

Ans. **Virtual Function** – Refer to Q.13.

Necessity of Virtual Function – Polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes. This necessiates the use of a single pointer variable to refer to the objects of different classes. Here, we use the pointer to base class to refer to all the desired objects. But a base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class. The compiler ignores the contents of the pointer and chooses the member function that matches the type of the pointer. Then virtual functions are used to achieve polymorphism virtual functions.

Q.15. What are virtual functions ? Create a base class named rectangle that contains data members length and width and member function Area ( ) and a constructor. From this class derive Box class with a property depth, an overridden function Area ( ) that returns the surface area of the box and a function called volume ( ). Include these classes in a working program. ***(R.G.P.V., Dec. 2011)***

Ans. **Virtual Function** – Refer to Q.13.

Program –

```
#include <iostream.h>
using namespace std;
class rectangle
{
    int length;
    int width;
    public :
        rectangle (int, int);
        int area( );
}
rectangle : : rectangle (int x, int y)
{
    length = x;
    width = y;
}
int rectangle : : area( )
{
    return (length * width);
}
class box : : public rectangle
```

```
{
    int depth;
    public:
        box (int d)
        {depth = d;}
    int area( )
    {
        return (2*(length*width + width*depth + depth*length));
    }
    int volume( )
    {
        return (length*width*depth);
    }
}
    int main( )
{
    rectangle γ (10, 10);
    box b (10);
    cout << γ.area( ) << "\n";
    cout << b.area( ) << "\n";
    cout << b.volume( ) <<"\n";
    return (0);
}
```

Q.16. What are virtual functions ? What is early binding and late binding and why is late binding preferred ? ***(R.G.P.V., Dec. 2010)***

Ans. **Virtual Functions** – Refer to Q.13.

Early and Late Binding – Refer to Q.5.

Consider for example, the following class definitions –

```
class X
    {
        int a;
        public:
                void show( ) {...}          // show( ) in base class
    };
class Y: public X
    {
        int b;
        public:
                void show( ) {...}          // show( ) in derived class
    };
```


The function is not overloaded and thus static binding does not apply as the prototype of show( ) is the same in both places. In this situation we may use the class resolution operator to specify the class, while invoking the functions with the derived class objects. It would be better if the suitable member function is invoked while the program is running.

Q.17. What are rules for virtual functions ?

Or

While creating virtual functions for implementing late binding what basic rules should be followed ? ***(R.G.P.V., June 2006)***

Ans. We should observe some basic rules that satisfy the compiler requirements –

(i) A virtual function can be ***friend*** of another class.

(ii) They are accessed by using object pointer.

(iii) Virtual destructor does exist but we cannot have virtual constructor.

(iv) Virtual function must be members of some class.

(v) Virtual function cannot be static members.

(vi) A virtual function in a base class must be defined, even though it may not be used.

(vii) If a virtual function is defined in the base class, it need not to be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

(viii) When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to next object.

(ix) The prototypes of base class version of virtual function and all the derived class versions must be identical. If two functions with same name have different prototypes, C++ considers them as overloaded function, and virtual function mechanism is ignored.

Q.18. What is pure virtual function ? Why do we need them and how are they defined ? ***(R.G.P.V., Dec. 2001)***

Ans. A ***pure virtual function*** is a function declared in a base class that has no definition relative to the base class. In such cases the compiler requires each derived class to either define the function or redeclare it as a pure virtual function.

Note that a class containing pure virtual function cannot be used to declare any objects of its own. Such classes are called abstract base classes.

A pure virtual function is declared as follows –

virtual return_type name (list of arguments) = 0;

Necessity of Pure Virtual Function – Some classes, such as class shape represent abstract concepts for which objects cannot exist. A shape makes sense only as the base of some class derived from it. This can be seen from the fact that it is not possible to provide sensible definitions for its virtual functions –

```
class shape {
    public:
    virtual void rotate(int) {error ("shape :: rotate");}
    // inelegant
    virtual void draw( ) {error ("shape : : draw");}
    .....
};
```

Trying to make a shape of this specified kind is silly but legal –

```
shape s;                    //silly "shapeless shape"
```

It is silly because every operation on s will result in an error.

A better alternative is to declare the virtual function of class shape to be pure virtual functions. A virtual function is "made pure" by initializer = 0 –

```
class shape {                   // abstract class
public:
    virtual void rotate(int) = 0;   // pure virtual function
    virtual void draw( ) = 0;       // pure virtual function
    ....
};
```

The equal sign here has nothing to do with assignment, the value 0 is not assigned to anything. The = 0 syntax is simply to tell the compiler that a function will be pure. A class with one or more pure virtual function is an abstract class, and no objects of that abstract class can be created –

```
shape s;          // error : variable of abstract class shape
```

An abstract class can be used only as an interface and as a base class for other classes.

When a class declaration contains a pure virtual function we cannot create an object of that class. Classes with pure virtual function exist solely to serve as base classes. For a class to be genuine abstract base class, it has to have at least one pure virtual function.

Abstract method may be declared to be 'pure virtual functions' in C++ any virtual function which is dynamically bound, can be made abstract by the following syntax –

virtual method_name( ) = 0;

This means that no implementation is provided for the method in this class, and has two implications –

(i) The method cannot be invoked by objects of derived classes unless an overriding implementation is specifically provided.

(ii) No objects of the class containing the abstract method may be instantiated.

Once we have placed a pure virtual function in the base class, then we must override it in all the derived classes from which we want to instantiate objects. If a class does not override the pure virtual function, then it becomes an abstract class itself, and we cannot instantiate objects from it. For consistency, we may want to make all the virtual functions in the base class pure.

Q.19. What are virtual functions ? What are pure virtual functions ? Discuss with the help of an example. ***(R.G.P.V., June 2012)***

Ans. **Virtual Function** – Refer to Q.13.

Pure Virtual Function – Refer to Q.18.

Q.20. Can a destructor be virtual ? Can a constructor be virtual ? Explain. ***(R.G.P.V., Dec. 2010)***

Ans. Destructors are declared as virtual, in contrast constructors cannot be virtual. Virtual destructors are controlled in the similar manner as virtual functions. Destructor of the derived class as well as destructors of all its base classes are invoked when a derived object pointed to by the base class pointer is deleted. The concept is illustrated by the following program –

```
#include <iostream.h>
#include <string.h>
class Father
{
    protected:
        char *father_name;
    public:
        Father(char *name)
        {
            father_name = new char[strlen(name) + 1];
            strcpy(father_name, name);
        }
    virtual ~ Father( )     //virtual destructors
        {
        delete father_name;
        cout << "~Father( ) is called" << endl;
        }
        virtual void display( )  //virtual function
        {
            cout << "Father's Name:" << father_name << endl;
        }
};
```

```
class Son:public Father
    {
        protected:
        char *son_name;
        public :
        Son(char *fname, char *sname):Father (fname)
        {
            son_name = new char[strlen(sname) + 1];
            strcpy(son_name, sname);
        }
        ~Son( )
        {
            delete son_name;
            cout << "~Son( ) is called" << endl;
        }
        void display( )
        {
            cout <<"Father's Name:"<<father_name<<endl;
            cout <<"Son's Name: "<<son_name<<endl;
        }
    };
    void main( )
    {
        Father *f;
        f = new Father ("Amitabh");
        cout << "f points to base object ..... "<<endl;
        f -> display( );
        delete f;
        f = new Son("Amitabh", "Abhishek");
        cout << "f points to derived object....."<<endl;
        f -> display( );
        delete f;
    }
```

In the above program, if the destructor is made as non-virtual destructor in the base class, only the base class's destructor is called when the object is deleted.

In main( ), the variable f is a pointer to base class Father. The statement

f = new Son("Amitabh", "Abhishek");

creates dynamic object of the class Son by allocating memory needed for its data members also. It is essential that memory allocated to object and its data members has to be released explicitly when the object pointed to by f goes out of scope.

When the destructor of the base class is not a virtual function in the normal case, the statement

delete f;

has deleted only the first string through the base class destructor, although in this case it also deletes the string, Amitabh through the derived class destructor. The base class destructor is declared as virtual and f actually addresses the Son's object and therefore, the destructors in the Son's class as well as the Father's class are called.

In the following situations virtual destructor is used –

(i) Destructors of a base class are declared as virtual functions. When a delete operations is performed on an object by a pointer or reference, the program will first invoke the object destructor rather than the destructor associated with the pointer or reference type.

(ii) A virtual destructor should be used when one class requires to delete object of a derived class that are addressed by the base pointers and call a base class destructor to free resources allocated to it.

Q.21. What is meant by overriding member function ? Explain containership. How does it differ from inheritance ? ***(R.G.P.V., June 2015, 2016)***

Ans. Overriding is the term used for redefining a method in a derived class, thus, providing specialized behaviour. When inheritance is used for specialization, a class extends its superclass in order to redefine some of the superclass methods, while leaving unchanged the rest of the methods it inherits. The inherit methods that are redefined are said to be overridden by the subclass. By overriding certain methods, the subclass specializes the behaviour of its superclass.

Consider, for example,

```
#include <iostream.h>
class BaseClass
{
    protected :
        int x;
    public :
        void setx (int x_in);
        void showx( );
};
void BaseClass : : setx (int x_in)
{
        x = x_in;
}
void BaseClass : : showx( )
{
        cout <<"base x =" << x << endl;
}
class DerivedClass : public BaseClass
```

```
{
    private :
        int y;
    public :
        void sety(int y_in);
        void showy( );
};
void DerivedClass : : sety (int y_in)
{
    y = y_in;
}
void DerivedClass : : showy( )
{
    cout <<"Derived y=" <<y<< endl;
}
```

Consider the base class method 'showx'. Whether it is used for objects of either the base or derived class, it will always display 'base x=' before the attribute value, even when the derived class is using the method.

In this program, objects of both the base and derived classes are instantiated and given values for their x attributes. Then the 'showx' method is called for both objects –

```
void main( )
{
    BaseClass base_object.
    DerivedClass derived_object;
    //set the x value for the base class object
        base_object.setx(8);
    //set the x value for the derived class object
        derived_object.setx (12);
    //display the x value for the both objects
        base_object.showx( );
        derived_object.showx( );
}
```

The output is as follows –

base x = 8

base x = 12

To overcome this problem, we can use the power of polymorphism to define a new 'showx' method for the derived class which will override the inherited version. Only the changes to the class definition have been included, other elements would remain unchanged.


```
class DerivedClass
{
    .....
    public :
        void showx( );
        .....
};
void DerivedClass : : showx( )
{
    cout <<"derived x=" <<x<<endl;
}
```

If we run program again, then we get as follows –

base x = 8

derived x = 12

Containership and Its Difference from Inheritance – Refer to Q.37 (Unit-III).

Q.22. Explain overriding. ***(R.G.P.V., Nov. 2018)***

Ans. Refer to Q.21.

Q.23. What is overriding ? What are the reasons of a subclass to override methods ? ***(R.G.P.V., Dec. 2013)***

***Ans.* Overriding** – Refer to Q.21.

There are several reasons for a subclass to override methods, of which the most common are as follows –

(i) To create a class some of whose behaviours are more specialized than those of its parent class. Here the new class represents a concept, that is a specialization of that of its parent class.

(ii) To create a class that is a variation of its parent class. The subclass reuses the elements of its superclass to simplify its own implementation.

(iii) To create a class that is more efficient than its superclass. The new class exhibits the same behaviour as its superclass but does so more efficiently, typically using less time or space.

(iv) To create a class that its more powerful, than its superclass. The new subclass is more powerful in the sense, that it does more with less. Greater power is achieved in one or both of the following ways –

(a) Some of the subclass's method override methods requiring stronger preconditions. This means that the subclass accomplishes as much as its superclass but requires less of its clients, or

(b) Some of the subclass's methods override methods promising weaker postconditions. This means that the subclass accomplishes more than its superclass, but without placing additional requirements on its clients.

Q.24. How methods can be redefined and used ? Give example.

Or

What is meant by overriding member functions ? Give proper example to explain the concept.

Or

Write short note on redefined methods. ***(R.G.P.V., June 2013)***

Ans. **Overriding Member Functions** – Refer to Q.21.

Redefinition takes a base class and redefines an existing method, implements it in a new way in order to change capability or performance. The inherited methods that are redefined are said to be overridden by the subclass. By overriding certain methods, the subclass specializes the behaviour of its superclass.

If a derived class fails to redefine a virtual function, the class will use the base class version of the function. If a derived class part of a long chain of derivations, it will use the most recently defined version of virtual function.

Suppose we create something like the following –

```
class Dwelling
{
    public:
    virtual void showperks(int a) const;
    ...
};
class Hovel : public Dwelling
{
    public:
    void showperks( );
    ...
};
```

This causes a problem. We may get a compiler warning similar to the following –

Warning : Hovel :: showperks(void) hides Dwelling :: showperks(int)

Or perhaps we won't get a warning. Either way, the code has the following implications,

```
Hovel trump;
trump.showperks( ); //valid
trump.showperks(s); //invalid
```

The new definition defines a showperks( ) that takes no arguments. Rather than resulting in two overloaded versions of the function, this redefinition hides the base class version that takes an int argument. In short, redefining inherited methods is not a variation of overloading. If we redefine a function in a derived class, it does not just override the base class declaration with the same function signature. Instead, it hides all base class methods of the same name, regardless of the arguments signatures.

This fact leads to a couple of rules of thumb. First, if we redefine an inherited method, make sure we match the original prototype exactly. One exception is that a return type that is reference or pointer to a base class can be replaced by a reference or pointer to the derived class. Second, if the base class declaration is overloaded, redefine all the base class versions in the derived class –

```
class Dwelling
{
    public:
        // three overloaded showperks()
        virtual void showperks(int a) const;
        virtual void showperks(double x) const;
        virtual void showperks() const;
        ...
};
class Hovel : public Dwelling
{
    public:
        // three redefined showperks()
        void showperks(int a) const;
        void showperks(double x) const;
        void showperks() const;
        ...
};
```

If we redefine just one version, the other two become hidden and cannot be used by objects of the derived class. The redefinition can simply call the base-class version if no change is needed.

Q.25. Explain overloading with example. ***(R.G.P.V., June 2014)***

Or

Write short note on overloading. ***(R.G.P.V., Nov. 2018)***

Ans. Function (and operators) can be "overloaded" in C++, i.e., the same identifier (or operator symbol) can be used in the same scope as the name of different functions (or operators). The overloading capability of C++ makes it more attractive as extensible programming language because it facilitates the use of identifiers and operator symbols that are natural to the application. For example, overloading allows conventional operator symbols to be used for the complex arithmetic operators which conform to mathematical conventions.

Overloaded functions (operators) with the same identifier (operator symbol) must have different "signatures". A signature consists of the function name (operator symbol) and an ordered sequence of parameter types. Two signatures are different if the function names (operator symbols) are different

or if the corresponding parameter types do not match. Note that a typedef type name matches its definition, an array type with a bound matches one without a bound, and an array type matches a pointer type provided the array elements and the object referenced by the pointer are of same type. Matching types are considered to be equivalent.

Q.26. What is operator overloading ? List some of the operators that can be overloaded and those which cannot be. ***(R.G.P.V., June 2005)***

Ans. Operator overloading is an important feature of C++, and it allows us to attach additional meaning to operators relative to a class. When an operator is overloaded, it gains an additional meaning relative to the class in which it is used. The mechanism of giving such special meanings to an operator is known as operator overloading.

The general form of an operator function is as follows –

```
return_type class_name : : operator symbol (argument list)
{
    Function body    //task defined
}
```

Where *return_type* is the type of value returned by the specified operation. Symbol means an operator symbol being overloaded. The word operator immediately after the scope resolution operator is the reserved word and qualifies the symbol used.

We can overload all the C++ operator listed in table 4.1, except the following-

(i) Dot operator (.)
(ii) Scope resolution operator (: :)
(iii) Conditional operator (? :)
(iv) Pointer-to-member operator (.*)
(v) Sizeof operator (sizeof)

Table 4.1 C++ Overloading Operators

Operator Category	Operators
Arithmetic	+, –, *, /, %
Bit-wise	&, |, ~, ^
Logical	&&, ||, !
Relational	>, <, = =, !=, <=, >=
Assignment or initialization	=
Arithmetic assignment	+=, –=, *=, /=, %=, &=, |=, ^=
Shift	<<, >>, <<<=, >>=
Unary	++, – –
Subscripting	[ ]
Function call	( )
Dereferencing	–>
Unary sign prefix	+, –
Allocate and free	new, delete

Q.27. What is operator overloading ? Can friend function be used to overload the assignment operator = ? Justify your answer. ***(R.G.P.V., Dec. 2003)***

***Ans.* Operator Overloading** – Refer to Q.26.

No, friend function cannot be used to overload the assignment operator =.

Though friend function can be used to overload operators but not all of them and assignment operator = is one such exception, because friend function will have only one argument for unary operators and two for binary operators while a member function has no argument for unary operators and only one for binary operators. This is because of the object used to invoke the member function is passed implicitly and therefore is available for the member function. While this is not the case with friend function, arguments may be passed either by value or by reference.

Q.28. What are the rules/limitations for overloading operators ?

Ans. Rules/limitations for overloading operators are as follows –

(i) Only existing operator can be overloaded.

(ii) The overloaded operator must have at least one operand, that is, the user defined type.

(iii) We cannot change basic meaning of an operator. That is, we cannot redefine the plus (+) operator to subtract one value from the other.

(iv) Overloaded operator follows the syntax rules of the original operators. They cannot be overridden.

(v) There are some operators, that cannot be overloaded.

(vi) We cannot use friend functions to overload certain operators.

For example,

= Assignment operator
( ) Function call operator
[ ] Subscripting operator
→ Class member access operator.

However member functions can be used to overload them.

(vii) Unary operators, overloaded by means of a member function, take no explicit argument and return no explict value, but those overloaded by means of a friend function take one reference argument.

(viii) Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.

(ix) When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.

(x) Binary arithmetic operators such as +, –, * and / must explicitly return a value. They must not attempt to change their own arguments.

Q.29. What are the limitations of overloaded operator functions ? Write a program in C++ to overload unary operator using friend function.
(R.G.P.V., June 2007)

Ans. **Limitations of Overloaded Operator Functions** – Refer to Q.28.

Program –

```
#include <iostream.h>
using namespace std;
class space
{
        int x;
        int y;
        int z;
    public :
        void getdata(int a, int b, int c);
        void display(void);
        friend void operator – (space & s);    // overload unary minus
                                               // using friend function
};
void space : : getdata(int a, int b, int c)
{
        x = a;
        y = b;
        z = c;
}
void space : : display(void)
{
        cout << x < < "  ";
        cout << y < < "  ";
        cout << z << "\n";
}
void space : : operator – (space & s)
{
        s.x = – s.x;
        s.y = – s.y;
        s.z = – s.z;
}
int main( )
{
        space S;
        S.getdata (1, –2, 3);
        cout << "S :";
        S.display( );
        –S;
        cout << "S :";
        S.display( );
        return 0;
}
```

Output –

```
S :    1    –2    3
S :   –1     2   –3
```


Q.30. What is operator overloading ? Write a distance class with data members for distance in metres, centimetres and millimetres. The class must be having overloaded operators + and – for addition and subtraction respectively. ***(R.G.P.V., June 2009, Dec. 2009, 2014)***

Or

Write a distance class with data members for distance in metres, centimetres and millimetres. The class must be having overloaded operators + and – for addition and subtraction respectively. ***(R.G.P.V., Dec. 2015)***

Ans. **Operator Overloading** – Refer to Q.26.

Program –

```
#include <iostream.h>
class distance
{
  float metres, centimetres, millimetres;
      public:
  distance (float m = 0.0, float cm = 0.0, float mm = 0.0)
  {
      metres = m;
      centimetres = cm;
      millimetres = mm;
  }
  void getvalue( );
  void showvalue( );
  distance operator + (distance d);
  distance operator – (distance d);
};
  void distance : : getvalue( )
  {
  cout <<"\n Enter the values";
  cin>>metres>>centimetres>>millimetres;
  }
  void distance : : showvalue( )
  {
  cout <<metres<<"m," <<centimetres<<"cm,"<<millimetres<<"mm";
  }
  distance distance : : operator + (distance d)
  {
  distance temp;
  temp.millimetres = millimetres + d.millimetres;
  temp.centimetres = 0;
  if (temp.millimetres > = 10)
  {
  temp.millimetres – = 10;
  temp.centimetres + +;
  }
```

```
temp.centimetres + = centimetres + d.centimetres;
temp.metres = 0;
if(temp.centimetres > = 100)
{
temp.centimetres – = 100;
temp.metres ++;
}
temp.metres + = metres + d.metres;
return temp;
}
distance distance : : operator – (distance d)
{
distance temp;
temp.millimetres = millimetres – d.millimetres;
temp.centimetres = centimetres – d.centimetres;
temp.metres = metres – d.metres;
return temp;}
void main( )
{
distance d1, d2, d3;
d1.getvalue( );
d2.getvalue( );
cout <<"\n value of d1 =" <<d1.showvalue( );
cout <<"\n value of d2 =" <<d2.showvalue( );
d3 = d1 + d2;
cout <<"value after addition =";
d3.showvalue( );
d3 = d1 – d2;
cout <<"value after subtraction =";
d3.showvalue( );
getch( )
}
```

Q.31. Explain operator overloading with the implementation of complex number and its numeric operations addition, subtraction, multiplication and division.
(R.G.P.V., June 2010)

Ans. **Operator Overloading** – Refer to Q.26.

Program –

```
#include <iostream.h>
class complex
{
      private:
      float x, y;
      public:
      complex( )
```



```
{
      x = y = 0;
}
void getdata( )  //read complex number
{
      cout << "Real Part ?";
      cin >> x;
      cout << "Imag Part ?";
      cin >> y;
}
void outdata (char *msg) //display complex number
{
      cout << endl << msg;
      cout << "(" << x;
      cout << ", " << y << ")";
}
      complex operator + (complex c2);
      complex operator – (complex c2);
      complex operator * (complex c2);
      complex operator / (complex c2);
};
//Addition of complex numbers
complex complex : : operator + (complex c2)
{
      complex temp;
      temp.x = x + c2.x;
      temp.y = y + c2.y;
      return(temp);
}
//Subtraction of complex numbers
complex complex : : operator – (complex c2)
{
      complex temp;
      temp.x = x – c2.x
      temp.y = y – c2.y;
      return(temp);
}
//Multiplication of complex numbers
complex complex : : operator * (complex c2)
{
      complex temp;
      temp.x = x * c2.x – y * c2.y;
      temp.y = x * c2.y + y * c2.x;
      return(temp);
}
```

MSInverter

```
//Division of complex numbers
complex complex : : operator / (complex c2)
{
    complex temp;
    float qt;
    qt = c2.x * c2.x + c2.y * c2.y;
    temp.x = (x * c2.x + y * c2.y)/qt;
    temp.y = (y * c2.x – x * c2.y)/qt;
    return(temp);
}
int main( )
{
    complex c1, c2, c3;
    cout <<"Enter a Complex Number c1.... "<<endl;
    c1.getdata( );
    cout << "Enter another Complex Number c2.... "<<endl;
    c2.getdata( );
    cout << "Entered complex numbers are ..... ";
    c1.outdata("c1 =");
    c2.outdata("c2 =");
    cout << endl << "Computational results are .....";
    c3 = c1 + c2;
    c3.outdata ("c3 = c1 + c2:");
    c3 = c1 – c2;
    c3.outdata("c3 = c1 – c2:");
    c3 = c1 * c2;
    c3.outdata("c3 = c1*c2:");
    c3 = c1/c2;
    c3.outdata("c3 = c1/c2:");
    return(0);
}
```

Q.32. Create a class FLOAT that contains one float data member. Overload all the four arithmetic operators so that they can operate on the objects of FLOAT. ***(R.G.P.V., June 2012)***

Ans. Refer to Q.31.

Q.33. Explain the concept of operator overloading. Illustrate operator overloading to concatenate strings. ***(R.G.P.V., June 2011)***

***Ans.* Operator Overloading** – Refer to Q.26.

Program –

```
#include <iostream.h>
  using namespace std;
  #include <string.h>                    // for strcpy( ), strcat( )
  #include <stdlib.h>                    // for exit( )
  class string
  {
      private :
          enum { SZ = 80};               // size of string objects
          char str[SZ];                  // holds a string
      public :
          string( )                      // constructor, no args
              { strcpy(str, " ");}
          string(char s[ ])              // constructor, one arg
          { strcpy(str, s); }
          void display( ) const          // display the string
          { cout << str; }
          string operator + (string ss) const     // add strings
          {
              string temp;
              if (strlen (str) + strlen(ss.str) < SZ)
              {
              strcpy(temp.str, str);
              strcat(temp.str, ss.str);
              }
              else
              {cout << "\n string overflow"; exit (1);}
              return temp;
          }
  };
  int main( )
```

MsInventer

```
{
    string s1 = "\nMerry christmas !";      // uses constructor 2
    string s2 = "Happy new year !";         // uses constructor 2
    string s3;                              // uses constructor 1
    s1.display( );                          // display strings
    s2.display( );
    s3.display( );
    s3 = s1 + s2;
                                            // add s2 to s1
    s3.display( );                          // assign to s3
    cout << endl;                           // display s3
    return 0;
}
```

Q.34. What is function overloading ? How polymorphism is achieved by function overloading ? ***(R.G.P.V., June 2005)***

Ans. Function overloading is a mechanism that helps us to get compile-time polymorphism. It means one interface and many methods.

When we overload the function, we can use the same function name to create functions that perform a variety of different tasks. This is possible because C++ permits the use of same name by two or more functions, provided the type of their arguments differs or number of their arguments differs or both differ.

A function can be overloaded by declaring and defining all the required versions. The compiler selects the correct version based on the number and type of arguments used at the time of function call.

Q.35. What is method overloading ? Give its advantages in object oriented programming. ***(R.G.P.V., June 2010)***

Ans. **Method Overloading** – Refer to Q.34.

Advantages of Method Overloading – The advantages of method overloading are as follows –

(i) It helps to make programs code more readable and understandable.

(ii) Debugging of code is very simple.

(iii) Programmer does not need to use different function name for the same operation.

(iv) Code can be easily maintain.

Q.36. What is method overloading ? What are the important points which should be taken care of while overloading methods ? ***(R.G.P.V., June 2009)***

Ans. **Method Overloading** – Refer to Q.34.

Important Points that should be Taken Care of while Overloading Methods – The following remarks can be made on function overloading –

(i) The use of more than one function with the same name, but having different actions should be avoided.

(ii) C++ does not permit overloading of functions differing only in their return value. The reason is that it is always the programmer's choice to inspect or ignore the return value of a function. For instance, the fragment

```
printf("Hello Guys!\n");
```

holds no information concerning the return value of the function print( ). Two functions printf( ), which would only differ in their return types and hence they are not distinguished by the compiler.

(iii) Function overloading can lead to surprises. For instance, imagine the usage of statements

```
show(0);
show(NULL);
```

Here, the zero could be interpreted as a NULL pointer to a char, i.e., a (char*) 0, or as an integer with the value zero. C++ will invoke the function expecting an integer argument, which might not the what the user expects.

Q.37. What do you mean by overloading of a function ? When do we use this concept ? ***(R.G.P.V., June 2013)***

Ans. **Overloading of a Function** – Refer to Q.34.

When we use this Concept – Function overloading is needed, because, by this, we can design a family of functions with one function name but with different argument list. The function would perform different operations depending on the argument list in the function call. The correct function to be called is determined by checking the number and type of the arguments but not on the function type.

Q.38. Write the example of function overloading.

Ans.

```
// Function volume( ) is overloaded three times
#include <iostream.h>
using namespace std;
// Declarations (prototypes)
int volume(int);
double volume(double, int);
long volume(long, int, int);
int main( )
{
      cout << volume(10) << "\n";
      cout << volume(2.5, 8) <<, "\n";
      cout << volume(100, 75, 15) << "\n";
      return 0;
}
// Function definitions
int volume(int s) // cube
{
return(s * s * s);
}
double volume(double r, int h) // cylinder
{
return(3.14519 * r * r * h);
}
long volume(long l, int b, int h) // rectangular box
{ return(l * b * h);
}
```

Q.39. Write a program to add 2 complex numbers using operator overloading. What are the operators that cannot be overloaded in C++? **(R.G.P.V., Dec. 2010)**

Ans. Program –

```
#include <iostream.h>
class complex
{
      private:
            float x, y;
      public:
            complex( )
            {
                  x = y = 0;
            }
```



```
            void getdata( )
            {
                  cout << "Real Part ?";
                  cin >> x;
                  cout << "Imaginary Part ?";
                  cin >> y;
            }
      void outdata(char *msg)
      {
      cout << endl << msg;
      cout << "(" << x;
      cout << "," << y << ")";
      }
      complex operator + (complex c2)
};
      complex complex : : operator + (complex c2)
      {
      complex temp;
      temp.x = x + c2.x;
      temp.y = y + c2.y;
      return(temp);
      }
      int main( )
      {
      complex c1, c2, c3;
      cout <<"Enter a complex number c1....."<<endl;
      c1.getdata( );
      cout<<"Enter a complex number c2...."<<endl;
      c2.getdata( );
      c3=c1 + c2;
      c3.outdata("c3 = c1 + c2:");
      }
```

Operators that cannot be Overloaded – Refer to Q.26.

Q.40. Using friend function write a program to overload == operator to compare two strings of class string. **(R.G.P.V., Dec. 2012)**

Ans.

```
#include <string.h>
#include <iostream.h>
class string
```

Msinverter

```
{
        int leng;
        char *p;
    public :
        string( ) {leng = 0; p = 0;}    // create null string
        string (const char *st);        // create string from arrays
        string (const string &st);      // copy constructor
    ~string( ) {delete p;}              // destructor
        friend int operator = = (const string &st, const string &tm);
        friend void show (const string st);
};
string : : string (const char *st)
{
    leng = strlen (st);
    p = new char [leng + 1];
    strcpy (p, st);
}
string : : string (const string &st)
{
    leng = st. leng;
    p = new char [leng + 1];
    strcpy (p, st. p);
}
//  overloading = = operator
int operator = = (const string &st, const string &tm)
{
    int a = strlen (st.p);
    int b = strlen (tm.p);
    if (a = = b) return (1);
    else return (0);
}
void show (const string st)
{
    cout < < st. p;
}
int main
{
    string s1 = "Hemant";
    string s2 = "Devesh";
    string t1, t2;
    t1 = s1;
    t2 = s2;
```



```
    cout <<"\nt1="; show (t1);
    cout <<"\nt2 ="; show (t2);
    if (t1 = t2)
    {
        show (t1);
        cout < < "equal to";
        show (t3);
    }
    else
    {
        show (t1);
        cout << "not equal to" ;
        show (t2);
    }
    return 0;
}
```

Q.41. Create a class called time that has separate int member data for hours, minutes and seconds. One constructor should initialize this to 0 and another should initialize it to fixed valued. A member function should display it in HH : MM : SS format. The final member function should add two objects of time passed as arguments.

Or

Write a program to create a class time containing data hours, minutes and seconds. Now overload binary + operator such that you can add two objects of the class time. Display the added object.

Or

Create a class TIME. Create T_1, T_2 and T_3 as objects of TIME. Write a C++ program to read time T_1 and T_2 as hh, mm and ss.

Where hh indicates hours, mm indicates minutes and ss indicates seconds. Assume hh, mm and ss all to be integers. Add these times T_1 and T_2 and store the result in time T_3. Print the value of T_3. **(R.G.P.V., June 2015)**

Ans.

```
#include <iostream.h>
class time
{
        int hr, min, sec;
    public :
        time( )
        {hr = 0; min = 0; sec = 0;}
        time (int h, int m, int s)
        {hr = h; min = m; sec = s;}
        void display( )
```

Msinventer

```
        {
            cout << "\n The time is \n";
            cout<<h<<"HH:"<<m<<"MM:"<<s<<"SS";
        }
        time operator + (time);
};

time time : : operator + (time t)
{
    time temp;
    temp.sec = sec + t.sec;
    temp.min = temp.sec/60;
    temp.sec = temp.sec % 60;
    temp.min = temp.min + min + t.min;
    temp.hr = temp.min/60;
    temp.min = temp.min % 60;
    temp.hr = temp.hr + hr + t.hr
    return(temp);
}
void main( )
{
    time T1 (6, 30, 45), T2(2, 50, 35), T3;
    cout << "\n The time of T1 \n";
    T1.display( );
    cout << "\n The time of T2 \n";
    T2.display( );
    T3 = T1 + T2;
    cout << "\n The time of T3 \n";
    T3.display( );
}
```

Q.42. Is operator overloading good or bad ? Comment. ***(R.G.P.V., Dec. 2004)***

Ans. Operator overloading is very useful (good). Operators such as the arithmetic operators +, –, * and / have certain fixed meaning to the compiler- they can be used with various numeric data types such as int and float. If we create new data types for use in our programs, then we can also use these symbols to mean operations on our own data types, even though the compiler does not automatically recognize them. This is made possible in object-oriented languages by overloading the meaning of an operator so that its behaviour can be polymorphic.

Operator overloading provides a flexible option for the creation of new definition for most of the C++ operators. We can almost create a new language

of our own by the creative use of the function and operator overloading techniques. We can overload all the C++ operators except few. The excluded operators are very few when compared to the large number of operators which qualify for the operator overloading definition.

Operator overloading extends the semantics of an operator but cannot change its syntax, the grammatical rules that govern its use. For example, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

Q.43. What do you mean by friend function ? ***(R.G.P.V., Dec. 2016)***

Ans. The private members cannot be accessed from outside the class. That is, a non-member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. When we want to access private data of a class by the function that is not a member function of that class. Then that function declaration should be preceded by the keyword *friend*. That function is called ***friend function***.

Thus, a friend which is not a member of a class but still has access to private data of a class. Friend functions are mainly useful in operator overloading.

We can declare this function as a *friend* of the class shown as follows –

```
class ABC
{
        ................
        ................
    public :
        ................
        ................
        friend void xyz(void); //declaration
};
```

A friend function can access the public, private, and protected members of the classes.

Q.44. Define friend function. Write a C++ code to demonstrate use of friend function.

Ans. **Friend Function** – Refer to Q.43.

Program – The following program illustrates the use of a friend function –

```
#include <iostream.h>
using namespace std;
class sample
```

```
{
    int p;
    int q;
    public:
        void setvalue( )
        {
            p = 25;
            q = 40;
        }
        friend float mean(sample s);
};
float mean(sample s)
{
    return float(s.p. + s.q.)/2.0;
}
int main
{
    sample 0; //object 0
    0.setvalue( );
    cout < < "Mean value =" << mean(0) << "\n";
    return 0;
}
```

Output –

Mean value = 32.5

Q.45. What are the merits and demerits of friend function ?

Ans. **Merits** – The merits of friend function are as follows –

(i) Friend functions can access the private data of other class.

(ii) It can be invoked like a normal function without the help of any object.

(iii) It can be used to alter the values of the private members of a class. But it should be used when necessary.

(iv) If friend function is called by reference, then the local copies of objects are not made and the called function directly works on the actual object used in the call.

(v) It can be declared either in the public or the private part of a class without affecting its meaning.

Demerits – The demerits of friend function are as follows –

(i) It cannot access the member names directly.

(ii) It is not in the scope of the class to which it has been declared as *friend*.

(iii) It cannot be called using the object of that class.

UNIT 5

STRINGS

Q.1. Discuss in brief string.

Ans. A string is an array of characters. Any group of characters defined between double quotation marks is a constant string. That is,

"Hello world"

Character strings are often used to build meaningful and readable programs. There are many operations that can be performed on character strings –

(i) Reading and writing strings. (ii) Combining strings together.
(iii) Copying one string to another. (iv) Comparing strings for equality.
(v) Extracting a portion of a string.

A string constant is a one dimensional array of characters terminated by a null ('\0'). For example,

char name [ ] = {'H', 'A', 'E', 'S', 'L', 'E', 'R', '\0'};

The terminating null is important because it is the only way the functions that work with string can know where a string ends. In fact, a string not terminated by a'\0', is not really a string, but merely a collection of characters. This string can be initialized as follows –

char name[ ] = "HAESLER";

Here, '\0' is not necessary. C inserts null automatically.

Q.2. Define a class string. Use overloaded = = operator to compare two strings with the help of program. ***(R.G.P.V., Dec. 2016)***

Ans. **Class String** – Refer to Q.1.

Example –

```
#include <iostream.h>
#include <string.h>
using namespace std;
int main( )
{
string s1 ("PQR");
string s2 ("XYZ");
```

```
string s3 = s1 + s2;
if (s1 ! = s2)
{
    cout << "s1 and s2 are not equal \n";
}
if (s1 > s2)
{
    cout << "s2 is less than s1 or s1 greater than s2" \n";
}
else
{
    cout << "s2 greater than s1 \n";
}
if (s3 = = s1 + s2)
{
    cout << "s2 is equal to s1 + s2 \n\n";
int x = s1.compare(s2);
}
if (x = = 0)
{
    cout << "s1 is equal to s2 \n";
}
}
```

Q.3. Explain the standard library string functions.

Ans. The C library supports a large number of string handling functions that can be used to carry out many of string manipulations.

There are two groups of the string functions defined in the header <string.h>. The first begins with ***str*** and second begins with ***mem***.

The most commonly used string functions are –

strcat( )	Concatenates two strings
strcmp( )	Compares two strings
strcpy( )	Copies one string over another
strlen( )	Finds the length of a string.

(i) strcat( ) Function – The strcat( ) function joins two strings together. The general form is –

strcat (string1, string2);

Here string1 and string2 both are character arrays. When strcat is executed string2 is appended to string1. It does so by removing the null character at the end of string1 and places string2 from there. The string at string2 remains unchanged.

Here is an example of strcat( ) –

```
main( )
{
    char source[ ]="folks!";
    char target[30]="Hello";
    strcat(target, source);
    printf("\n source string=%s", source);
    printf("\n target string=%s", target);
}
```

The output will be –

source string = folks!
target string=Hellofolks!

(ii) strcmp( ) Function – This function compares two strings to find that whether they are same or not. The two strings are compared character by character until there is a mismatch or end of one of the strings, is reached, whichever occurs first. If two strings are identical strcmp( ) returns a zero value. If they are not; it returns the numeric difference between ASCII values of first nonmatching pairs of characters. The general form is –

strcmp(string1, string2);

Here, string1 and string2 may be string variables or string constants.

The example of strcmp( ) function is as follows –

```
main( )
{
    char string1[ ] = "Jerry";
    char string2[ ] = "Ferry";
    int i,j,k;
    i = strcmp(string1, "Jerry");
    j = strcmp(string1, string2);
    k = strcmp(string1, "Jerry boy");
    printf("\n%d%d%d", i,j,k);
}
```

The output will be –

0 4 –32

In the first call, two strings are identical and returned value is zero. In second both strings do not match, the result is 4, which is the numeric difference between ASCII value of J and ASCII value of F. In the third call "Jerry" does not match with "Jerry boy". So the returned value is –32 due to the value of null character minus the ASCII value of space.

(iii) strcpy( ) Function – This function copies the contents of one string into another. The base addresses of the source and the target strings should be supplied to this function.

This function works almost like string assignment operator. The general form is –

strcpy(string1, string2);

It assigns the contents of string2 to string1. String2 may be a character array variable or a string constant.

For example –

```
main( )
{
        char source[ ]="sayonara";
        char target[20];
        strcpy(target, source);
        printf("\nsource string = %s", source);
        printf("\ntarget string = %s", target);
}
```

The output will be –

```
        source string = sayonara
        target string = sayonara
```

(iv) strlen( ) Function – This function counts the number of characters present in a string. The general form is –

n = strlen(string);

where n is an integer variable which receives the value of the length of the string. The argument may be a string constant. The counting ends at the first null character.

Here, the example of strlen( ) function is as follows –

```
main( )
{
    char arr[ ]= "Bamboozled";
    int len1, len2;
    len1 = strlen(arr);
    len2 = strlen("Humpty Dumpty");
    printf("\nstring = %s length = %d", arr, len1);
    printf("\nstring = %s length = %d", "Humpty Dumpty", len2);
}
```

The output will be –

```
string = Bamboozled length = 10
string = Humpty Dumpty length = 13
```

In the first call to the function, we are passing the base address of the string and function returns the length of the string, it does not count '\0'. In second call is passed the address of the string and not the string itself.

Q.4. How C++ String class better to C String class.

Ans. In C++, you no longer need to worry about creating an array of the right size to hold string variables. The String class allows the use of overloaded operators. The String class assumes all the responsibility for memory management. The C++ String class is more efficient and safer to use than C String class.

Q.5. How can you define and assign string objects ?

Ans. A string object can be define in many different ways. You can use a constructor with no arguments, creating an empty string. You can also use a one argument constructor, where the argument is a C string constant i.e., character delimited by double quotes. Object of string class can be assigned to one another with a simple assignment operator in String class. The program given below show how to define and assign string objects –

```
#include <iostream.h>
#include <string.h>
int main( )
{
    String a1("Honesty") ;              //initialize
String a2 = "Policy";                   //initialize
String a3;
    a3 = a1                             //assign
cout <<"a3 =" << a3 << endl;
a3 = a3 + "is best";                    //concatenate
a3 += a2;                               //concatenate
cout << "a3 = "<<a3<<endl;
a1.swap(a2);
cout <<a1<< "is" << a2 << endl;
return 0;
}
```

The first three lines of code show three different ways to define object. The first two initialize strings, and the third creates an empty string variable. The next line shows the assignment operator. The overload + operator concatenates one string object with another. The statement –

a3 = a3 + "is best";

places the string "Honesty is best" in the variable 93. You can also use the += operator to append a string to the end of an existing string. The statement –

a3 += a2;

appends a2, which is "Policy", to the end of a3, producing the string "Honesty is best Policy" and assigned it to a3. The member function swap( ) exchanges the values of two string objects and displays the output "Policy is Honest".

Q.6. Explain array of strings.

Ans. A list of names can be treated as a table of strings and a two-dimensional character array can be used to store the entire list. For example, a character array student [30] [15] may be used to store a list of 30 names, each

of length not more than 15 characters. A table of five cities is shown below –

C	h	a	n	d	i	g	a	r	h
M	a	d	r	a	s				
A	h	m	e	d	a	b	a	d	
H	y	d	e	r	a	b	a	d	
B	o	m	b	a	y				

This table can be easily stored in a character array city by using the following declaration –

```
char city [ ] [ ]
    {
        "Chandigarh",
        "Madras",
        "Ahmedabad",
        "Hyderabad",
        "Bombay"
    };
```

To access the name of the i^{th} city in the list, we write

city[i–1]

and therefore city[0] refers to "Chandigarh", city [1] refer to "Madras" and so on. This shows that once an array is declared as two-dimensional, it can be used like a one-dimensional array in further manipulation. That is, the table can be treated as a column of strings.

Q.7. What is the use of string buffer classes ?

Ans. String buffer is used to create strings of flexible size which can be modified in length and content. A new string can be appended to the end of string and new characters and substrings can be inserted in the middle or at require position between string.

Table 5.1 String Buffer Method for String Manipulation

S.No.	Method	Action
(i)	s1.setChartAt(n,'A')	Replace the nth character to A.
(ii)	s1.append(s3)	Appends the s3 string to the end of s1 string
(iii)	s1.insert(n, s3)	Inserts the string s3 at the string s1 nth position.
(iv)	s_1.setLength(n)	Sets the length of the string s1 to n. If n<s1. length ( ) s1 is truncated. If n > s1. length ( ) zeros are added to s1.

Q.8. Give the list of commonly used string methods.

Ans. The string class has multiple string methods which is used to perform multiple type of string manipulation tasks.

Table 5.2 String Methods

S.No.	Method	Action
(i)	s1 = s.toLowerCase;	Converts the string s in lower case
(ii)	s1 = s.toUpperCase;	Converts the string s in upper case
(iii)	s1 = s.replace ('X', 'R');	Modify all X with R
(iv)	s1 = s.trim( );	Delete all white spaces at the starting and ending of string s
(v)	String.ValueOf(Parameter)	Changes the parameter value to string representation
(vi)	s.indexOf ('A', n)	A is place after nth position in the string.
(vii)	s.indexOf ('A')	A are place at the first position in the string.
(viii)	q.toString( )	Creates a string representation of the object q
(ix)	String.ValueOf(q)	Creates a string object of the parameter q
(x)	s.substring(n, z)	Provided substring beginning from n^{th}, character up to z^{th} (not involving z^{th})
(xi)	s.substring(n)	Provided substring beginning from n^{th} character
(xii)	s.concat(s1)	Concatenates s and s1
(xiii)	s.compareTo(s1)	If s = s1, returns zero If s < s1, returns negative If s > s1, returns positive
(xiv)	s.length( )	Returns the length (number of char) of string s.
(xv)	s.chartAt(n)	Provides nth character of s
(xvi)	s.equals(s1)	If s = s1, returns true
(xvii)	s.equalsIgnoreCase (s1)	If s = s1, returns true, ignoring the case of characters

Q.9. Write generally used string constructors.

Ans. The generally used string constructors are shown in table 5.3.

Table 5.3 String Constructors

S.No.	Method	Action
(i)	String(const string & str);	It is used to create a string object from other string object.
(ii)	String (const char *str);	It is used to create a string object from a null-terminated string.
(iii)	String( );	It is used to create an empty string.

Q.10. Give the functions which is supported by the string class.

Ans. There are several functions which is supported by string class –

(i) ***append( )*** – It is used to append one string to another string.

(ii) ***assign( )*** – It is used to assign a partial string.

(iii) ***swap( )*** – It is used to swaps given string and invoking string.

(iv) ***size( )*** – It returns the number of characters in the given string.

(v) ***resize( )*** – It is used to modified the size of string as defined.

(vi) ***replace( )*** – It is used to change defined characters with a provided string.

(vii) ***insert( )*** – It is used to inserts characters at given position.

(viii) ***max_size( )*** – It is used to provide the maximum size of a string object in a give system.

(ix) ***find( )*** – It is used to searches for the occurrence of a define substring.

(x) ***erase( )*** – It is used to delete define character.

(xi) ***end( )*** – It is used to returns a reference to the last of a string.

(xii) ***empty( )*** – It is used to returns true if string is empty else false.

(xiii) ***capacity( )*** – It is used to know the capacity of provided string.

(xiv) ***at( )*** – It is used to obtains the character from specified location in a string.

(xv) ***begin( )*** – It is used to returns a reference to the beginning of the string.

Q.11. How can we create string objects ?

Ans. There are multiple way to create string objects –

```
string s;            // By a constructor with no argument
string s("xyz");     // By a constructor with one-argument
s = s1;              // String objects assigned
s2 = "xyz" + s1      // Concatenating strings
cin >> s1;           // Assigning through keyboard
getline (cin, s);    // Assigning through keyboard a line of text
```

Two strings objects can be concatenates by the overloaded + operator. + = operator is also used to append a string to the end of a string. For example -

```
s2 + = s1;        // s2 = s2 + s1
s2 + = "abc";     // s2 = s2 + "abc"
```

To manage input and output of string object, the operators << and >> are overloaded. For example –

```
cin >> s1;           // Input to string object
cout << s1;          // Represent the contents of 1
getline (cin, s1);   // Reads embedded blanks
```


Q.12. How can we manipulate string object ?

Ans. There are several way to manipulate content of string object like insert( ), replace( ), erase( ), append( ).

Program –

```
#include <iostream.h>
#include <string.h>
using namespace std;
int main ( )
{
    string s1 ("01234");
    string s2 ("pqrst");
    cout << "Strings are: \n";
    cout << "S1: " << s1 << "\n";
    cout << "S2: " << s2 << "\n\n";
    cout << "Put S2 inside S1 \n";
    s1.insert (3, s2);
    cout << "Now S1 is: " << s1 << "\n\n";
    cout << "Characters from S1 \n";
    s1.erase 3, 4);
    cout << "Now S1 is: " << s1 << "\n\n";
    cout << "Replace Middle 3 Characters in S2 with S1 \n";
    s2.replace (1, 3, s1);
    cout << "Now S2 is: " << s2 << "\n";
    return 0;
}
```

Output –

```
Original Strings are :
S1 : 01234
S2 : pqrst
Put S2 inside S1
Now S1 is 0123pqrst4
delete 5 Characters from S1
Now S1 is : 01234
Replace Middle 3 Characters in S2 with S1
Now S2 is : P01234t
```

Q.13. Give the list of operator for string object.

Ans. The list of operator for string objects are shown in table 5.4.

Table 5.4 String Objects Operators

S.No.	*Operator Symbol*	*Operator Name*
(i)	=	Assignment
(ii)	+	Concatenation
(iii)	+ =	Concatenation assignment
(iv)	= =	Equality
(v)	! =	Inequality
(vi)	<	Less than

(vii)	<=	Less than or equal
(viii)	>	Greater than
(ix)	>=	Greater than or equal
(x)	[ ]	Subscription
(xi)	<<	Output
(xii)	>>	Input

Q.14. How can we access characters in strings ?

Ans. There are several way to access characters in string. The string class uses following method for this purpose –

(i) ***at( )*** – It is used to get a character

(ii) ***Substr( )*** – To retrieving substring like 'abc'

(iii) ***find( )*** – It is used to search define substring

(iv) ***find_first_of( )*** – It is used to search the place of first occurrence of the define character (s)

(v) ***find_last_of( )*** – It is used to search the place of last occurrence of the define character (s)

EXCEPTIONAL HANDLING

Q.15. What is exception ?

Ans. An exception is an abnormal condition that arises in a code sequence at run-time. In other words, exception is a run-time error. When the Java encounters an error such as dividing an integer by zero, it creates an exception object and throws it. If an exception object is not caught and handled properly, the interpreter will display an error message and terminate the program.

Q.16. Explain the exception types.

Ans. All exception types are subclasses of the built in class *Throwable*. Thus, *Throwable* is at the top of the exception class hierarchy. Immediately below *Throwable* are two subclasses that partition exceptions into two distinct branches. One branch is headed by *Exception*. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of *Exception* called *RuntimeException*. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by *Error*, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type *Error* are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such as error.

Fig. 5.1 shows a partial class hierarchy for exceptions.

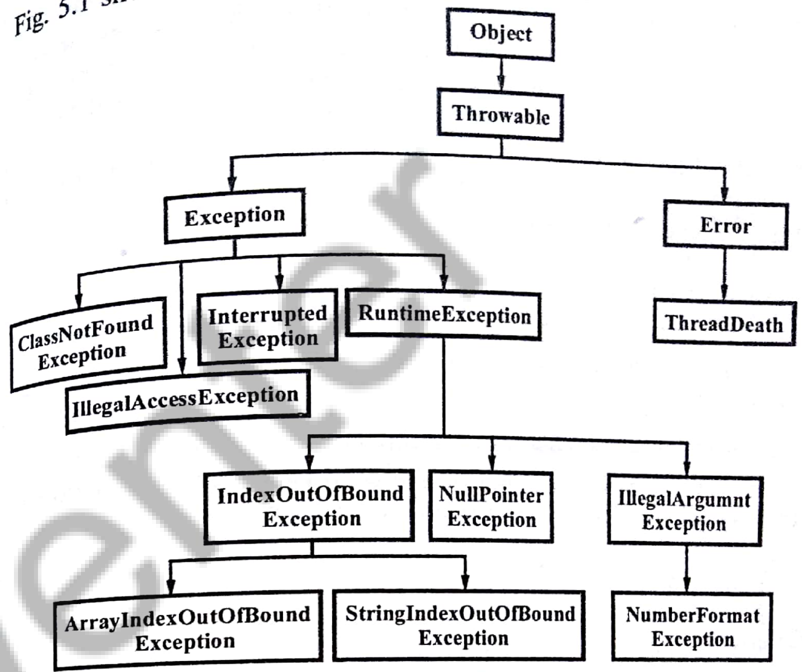


Fig. 5.1 Exception Hierarchy

Q.17. List different exceptions defined in Java.

Or

List some of the most common types of exceptions that might occur in Java. Give example. ***(R.G.P.V., Dec. 2015)***

Ans. Some common exceptions are listed in table 5.5.

Table 5.5 Common Java Exceptions

Exception Type	*Cause of Exception*
ArithmeticException	Caused by math errors such as division by zero
ArrayIndexOutOfBoundException	Caused by bad array indexes
ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
FileNotFoundException	Caused by an attempt to access a nonexistent file
IOException	Caused by general I/O failures, such as inability to read from a file
NullPointerException	Caused by referencing a null object
NumberFormatException	Caused when a conversion between strings and number fails

OutOfMemoryException	Caused when there's not enough memory to allocate a new object
SecurityException	Caused when an applet tries to perform an action not allowed by the browser's security setting
StackOverFlowException	Caused when the system runs out of stack space
StringIndexOutOfBoundsException	Caused when a program attempts to access a nonexistent character position in a string

Q.18. Explain advantage of exception. How exception sub classes are created ? ***(R.G.P.V., Dec. 2014)***

Ans. The main advantage of exception handling is to maintain the normal flow of application. Exception normally disrupts the normal flow of the application that is why we use exception handling. For example,

Statement 1;
Statement 2;
Statement 3;
Statement 4;
Statement 5;

Let these five statements in your program and there occurs an exception at statement 3, rest of the code will not be executed. If we perform exception handling, rest of the statement will be executed.

Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do – just define a subclass of exception. Your subclasses do not need to actually implement anything it is their existence in the type system that allows you to use them as exceptions. The exception class does not define any methods of its own. It does, of course, inherit those methods provided by Throwable. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them.

Q.19. Write short note on exception handling in Java. ***(R.G.P.V., June 2016)***

Ans. If the exception is not caught and handled properly, the interpreter will display an error message and will terminate the program. If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as ***exception handling.***

Q.20. What is the purpose of exception handling ?

Ans. The purpose of exception handling mechanism is to provide a means to detect and report an exceptional circumstance so that appropriate action can be taken. The mechanism suggests incorporation of a separate error handling code that performs the following tasks –

(i) Find the problem (*Hit* the exception)
(ii) Inform that an error has occurred (*Throw* the exception)
(iii) Receive the error information (*Catch* the exception)
(iv) Take corrective actions (*Handle* the exception)

The error handling code basically consists of two segments, one to detect errors and to throw exceptions and to take appropriate actions.

Q.21. Explain the basic concept of exception handling using try and catch statements.

Ans. The basic concepts of exception handling are throwing an exception and catching it. This is shown in fig. 5.2.

Java uses a keyword *try* to preface a block of code that is likely to cause an error condition and "throw" an exception. A catch block defined by the keyword *catch* "catches" the exception *"thrown"* by the try block and handles it appropriately. The catch block is added immediately after the try block. The following example shows the use of simple try and catch statements –

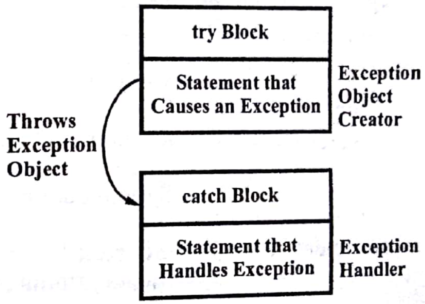


Fig. 5.2 Exception Handling Mechanism

```
.............
.............
try
{
    statement;          //generates an exception
}
catch (Exception-type e)
{
    statement;          //processes the exception;
}
............
............
```

The *try* block can have one or more statements that could generate an exception. If any one statement generates an exception, the remaining statements in the block are skipped and execution jumps to the catch block that is placed next to the try block.

The *catch* block too can have one or more statements that are necessary to process the exception. Every *try* statement should be followed by at least one *catch* statement, otherwise compilation error will occur.

The catch statement is passed a single parameter, which is reference to the exception object thrown. If the catch parameter matches with the type of exception object, then the exception is caught and statements in the catch block will be executed. Otherwise, the exception is not caught and the default exception handler will cause the execution to terminate.

A program shown below, the use of try and catch blocks to handle an arithmetic exception –

```
class error
{
    public static void main(String args[ ])
    {
        int a = 4;
        int b = 2;
        int c = 2;
        int x, y;
        try
        {
            x = a/(b – c);      //Exception here
        }
        catch(ArithmeticException e)
        {
            System.out.println("Division by zero");
        }
        y = a/(b + c);
        System.out.println("y=" +y);
    }
}
```

Output –

Division by zero
y = 1

Q.22. Describe the concept of stack based execution and exception propagation.

Ans. A stack is a special kind of list (FILO–first in last out). Think of it like stacking pennies one at a time. The first one that you put down will be the last one that you pick up when you are unstacking the pennies. Java maintains a stack of method calls to track the program flow from method to method. As an example, consider three methods, main( ), method1( ) and method2( ), where method1( ) is called from main( ) and method2( ) is called from method1( ). The stack starts out with the main( ) method. When method1( ) is called from the main( ) method, method1( ) goes to the top of the stack (adding to a stack is called a push). When method2( ) is called from method1( ) method2( ) is pushed onto the stack. When method2( ) completes, it gets removed from the stack (removing from a stack is called a pop). At this point, the top of the stack is method1( ). This make sense. Because method2( ) is called from method1( ), program control returns to method1( ) when method2( ) returns. When method1( ) returns, it get popped from the stack, leaving only main( ). This is how stack trace works and the execution by using stack is executed.

The compiler strictly enforces the throws specifiers. If you call a method that throws a checked exception, you must either handle it or pass it on.

Which of the two is better ? As a general rule, you should catch those exceptions that you know how to handle and propagate those that you do not know how to handle. When you propagate an exception may be thrown.

Look at the Java API documentation to see what method which exceptions. Then decide weather you should handle them or add them to throw list. There is nothing embrassing about the latter choice. It is better to direct an exception to a competent handler than to squelch it.

Please keep in mind that there is one exception to this rule if you are writing a method that overrides a superclass method throws no exceptions (such as paintComponent in JComponent), then you must catch each checked exception in the method's code. You are not allowed to add more throws specifiers to a subclass method that are present in the superclass method.

Q.23. Explain the following terms –

(i) Throw an exception (ii) Catching an exception
(iii) Try and catch block.

(R.G.P.V., Dec. 2013)

Or

Explain the following terms –

(i) Throw an exception (ii) Catching an exception.

(R.G.P.V., June 2015)

Ans. (i) Throw an Exception – When an exception is detected, it is thrown using a throw statement in the try block. Exceptions are thrown by functions are invoked from within the try block. The point at which the throw is executed is called the throw point.

Once an exception is thrown to the catch block, control cannot return to the throw point as shown in fig. 5.3.

The general form of code for this kind of relationship is as

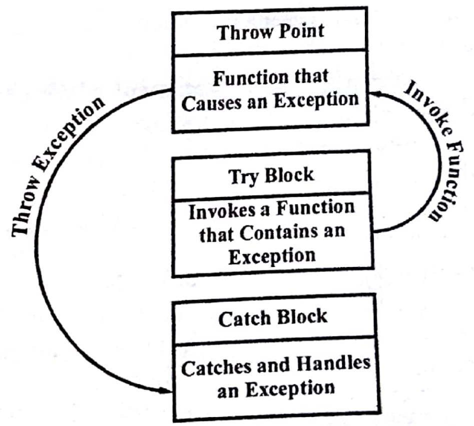


Fig. 5.3 Throwing an Exception

follows –

```
type function(arg list) // Function with exception
{
    ................
    ................
    throw(object) // Throws exception
    ................
    ................
}
    ................
    ................
try
{
    ................
    ................ // Invoke function here
}
catch(type arg) // catches exception
{
    ................
    ................ // Handles exception here
}
    ................
```

(ii) Catching an Exception – A catch block looks like a function definition and is of the form –

```
catch(type arg)
{
        //Statements for managing exceptions
}
```

The catch statement catches an exception whose type matches with the type of catch argument. When it is caught, the code in the catch block is executed.

Multiple Catch Statement – We can associate more than one catch statements with a try as follows –

```
try
{
        // try block
}
catch(type1 arg)
{
        // catch block1
}
catch(type2 arg)
```



```
{       // catch block2
}
................
................
catch(typeN arg)
{       // catch blockN
}
```

When an exception is thrown, the exception handlers are searched in order for an appropriate match. It is possible that arguments of several catch statements match the type of an execution. In such cases, the first handler that matches the exception type is executed.

Catch all Exceptions – We can force a catch statement to catch all exceptions instead of a certain type alone. This could be achieved by defining the catch statement as follows –

```
catch(.......)
{
    //statements for processing
    //all exceptions
}
```

It may be a good idea to use the catch(......) as a default statement along with other catch handlers so that it can catch all those exceptions which are not handled explicitly.

Note that catch(.....) should be placed last in the list of handlers. Placing it before other catch blocks would prevent those blocks from catching exceptions.

(iii) Try and Catch Block – Try block is a block of statements, which may generate exceptions. While a ***catch block*** catches the exception thrown by the throw statement in the try block and handles it appropriately. Fig. 5.4 shows try and catch block operations.

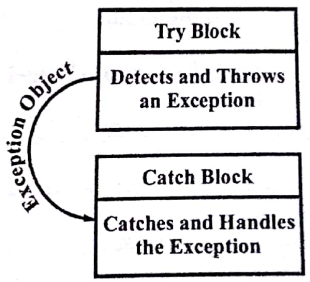


Fig. 5.4 Try and Catch Block

The catch block which catches an exception must immediately follow the try block that throws the exception. The general form of these two blocks are as follows –

```
................
............. ...
try
```

Msinventer

```
{ ..................
    throw exception.      // blocks of statements
    ..................    // which detects and
    ..................    // throws an exception
}
catch(type arg)
{   ..................    // block of statements that
    ..................    // handles the exception
}
    ..................
    ..................
```

When the try throws an exception, the program control leaves the try block and enters in the catch block. Note, the exceptions are objects used to transmit information about the problem. If the type of object thrown matches the arg type in the catch statement, then the catch block is executed for handling the exception. If they do not match, the program is aborted by abort( ) function, that is called by default.

When no exception is detected and thrown, the control goes to the statement immediately after the catch block, and catch block is skipped.

Q.24. How do we use multiple catch block ?

Or

Create a try block that is likely to generate three types of exception and then incorporate necessary catch blocks to catch and handle them appropriately.

Ans. It is possible to have more than one catch statements in the catch block as illustrated below –

```
..........
..........
try
{
    statement;                  //generates an exception
}
catch(Exception-Type-1 e)
{
    statement;                  //processes exception type1
}
catch(Exception-Type-2 e)
{
    statement;                  //processes exception type2
}
    :
    :
catch(Exception-Type-N e)
{
    statement;                  //processes exception type N
}
..........
..........
```


When an exception in a try block is generated, the Java treats the multiple catch statements like cases in a switch statement. The first statement whose parameter matches with the exception object will be executed, and the remaining statements will skipped.

The following program traps different exception types –

```
class error
{
    public static void main(String args[ ])
    {
        int a[ ] = {5, 10};
        int b = 5;
        try
        {
            int x = a[2]/b – a[1];
        }
        catch(ArithmeticException e)
        {
            System.out.println("Division by zero");
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Array index error");
        }
        catch(ArrayStoreException e)
        {
            System.out.println("Wrong data type");
        }
        int y = a[1]/a [0];
        System.out.println("y=" +y);
    }
}
```

Output –

```
array index error
y = 2
```

Q.25. What is finally block ? When and how it is used ? Give a suitable example.

Ans. When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception handling mechanism. The *finally* keyword is designed to address this contingency.

Finally creates a block of code that will be executed after a *try/catch* block has completed and before the code following the *try/catch* block. The *finally* block will

execute whether or not an exception is thrown. If an exception is thrown, the *finally* block will execute even if no *catch* statement matches the exception. Any time a method is about to return to the caller from inside a *try/catch* block, through an uncaught exception or an explicit return statement, the *finally* clause is also executed just before the method returns. The *finally* clause is optional. However, each *try* statement requires at least one *catch* or a *finally* clause.

An example program is shown below that shows three methods that exit in various ways, none without executing their *finally* clauses –

```
class FinallyDemo {
    static void procA( ) {
    try {
    System.out.println("inside procA");
    throw new RuntimeException("demo");
    } finally {
        System.out.println("procA's finally");
    }
}
    static void procB( ) {
        try {
            System.out.println("inside procB");
            return;
        }   finally {
            System.out.println("procB's finally");
        }
    }
static void procC( ) {
    try {
        System.out.println("inside procC");
    }   finally {
        System.out.println("procC's finally");
    }
}
public static void main(String args[ ]) {
    try {
        procA( );
    }   catch(Exception e) {
        System.out.println("Exception caught");
    }
        procB( );
        procC( );
    }
}
```

Here, *procA()* prematurely breaks out of the *try* by throwing an exception. The *finally* clause is executed on the way out. *procB( )'s try* statement is exited through a *return* statement. The *finally* clause is executed before *procB()* returns. In *procC()*, the *try* statement executes normally, without error. However, the finally block is still executed.

Output –

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```


Q.26. Explain throw statement with suitable example.

Ans. It is possible for your program to throw an exception explicitly, using the *throw* statement. The general form of *throw* is shown below –

```
throw ThrowableInstance;
```

Here, *ThrowableInstance* must be an object of type *Throwable* or a subclass of *Throwable*. Primitive types, such as *int* or *char*, as well as non-Throwable classes, such as *String* and *Object*, cannot be used as exceptions. There are two ways you can obtain a throwable object – using a parameter in a *catch* clause, or creating one with the *new* operator.

The flow of execution stops immediately after the **throw** statement. Any subsequent statements are not executed. The nearest enclosing *try* block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing *try* statement is inspected, and so on. If no matching *catch* is found, then the default exception handler halts the program and prints the stack trace.

An example program is shown below that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler –

```
class ThrowDemo {
    static void demoproc( ) {
    try {
    throw new NullPointerException("demo");
}
catch(NullPointerException e) {
        System.out.println("Caught inside demoproc.");
        throw e;        //rethrow the exception
    }
}

    public static void main(String args[ ]) {
        try {
            demoproc( );
        }   catch(NullPointerException e) {
            System.out.println("Recaught :" +e);
        }
    }
}
```

MSinfer

The above program gets two chances to deal with the same error. First, *main()* sets up an exception context and then calls *demoproc()*. The *demoproc()* method then sets up another exception handling context and immediately throws a new instance of *NullPointerException*, which is caught on the next line. The exception is then rethrown.

Output –

Caught inside demoproc.
Recaught: Java.lang.NullPointerException:demo

Q.27. Explain throws clause with a suitable example.

Ans. If a method is capable of causing an exception that it does not handle, it must specify this behaviour so that callers of the method can guard themselves against that exception. You do this by including a *throws* clause in the method's declaration. A *throws* clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type *Error* or *RuntimeException*, or any of their subclasses. All other exceptions that a method can throw must be declared in the *throws* clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a *throws* clause –

```
type method-name (parameter-list) throws exception-list
{
    ............
    ............            //body of the method
}
```

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a *throws* clause to declare this fact, the program will not compile –

```
//This program contains an error and will not compile
class ThrowsDemo
{
    static void throwOne( )
    {
        System.out.Println("Inside throwOne.");
        throw new IllegalAccessException("demo");
    }

        public static void main(String args[ ]) {
        throwOne( );
    }
}
```


To make this example compile, you need to make two changes. First, you need to declare that *throwOne()* throws *IllegalAccessException*. Second, *main()* must define a *try/catch* statement that catches this exception.

```
//This is now correct.
class ThrowsDemo
{
    static void throwOne( ) throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");

}
public static void main(String args[ ]) {
        try {
        throwOne( );
    }   catch(IllegalAccessException e) {
        System.out.println("Caught" + e); }
    }
}
```

Output –

Inside throwOne
caught java.lang.IllegalAccessException:Demo

Q.28. Write a program to demonstrate "IndexOutOfBoundsException".

Ans.

```
class Exc
{   public static void main(String args[ ])
    {
        try
        {
        int a = args.length;
        System.out.println("a =" +a);
        int b = 42/a;
        int c[ ] = { 1 };
        c[42] = 99;
        }
        catch(ArithmeticException e)
        {
            System.out.println("Divide by 0:" +e);
        }
        catch(ArrayIndexOutOfBoundsException e)
        {
        System.out.println("Array index oob :" +e);
        }
    }
}
```

This program will cause a division-by-zero exception if it is started with no command line arguments, since *a* will equal zero. It will survive the division if you provide a command-line argument, setting *a* to something larger than

MSinverter

zero. But it will cause an ArrayIndexOutOfBoundsException, since the int array *c* has a length of 1, yet the program attempts to assign a value to c[42].

Output –

```
C:\>java Exc
a = 0
Divide by 0: java.lang.ArithmeticException:/by zero
After try/catch blocks.
    C: \ > java Exc TestArg
    a = 1
    Array index oob : java.lang.ArrayIndexOutOfBoundsException : 42.
    After try/catch blocks.
```

Q.29. Write a class that keeps a running total of all characters passed to it (one at a time) and throws an exception if it is passed a non-alphabetic character.

Ans.

```
class A
{
    public static void main(String args[ ])
    {
        int count = 0;
        char c = 'y';
        while (c = = 'y')
        {
            try
            {
                System.out.println("Enter a character");
                str = args [i];
            }
            catch (ArrayStoreException e)
            {
                System.out.println("You entered a non-alphabetic character");
                continue;
            }
                count = count +1;
                System.out.println("You want to enter another character");
            }
            System.out.println("Total no. of characters passed" +count);
        }
    }
```

Q.30. What are assertions and how does the Java do enabling and disabling of assertions ?

Ans. When implementing and debugging a class, it is sometimes useful to state conditions that should be true at particular point in a method. These

conditions, called *assertions,* help ensure a program's validity by catching potential bugs and identifying possible logic errors during development. Preconditions and postconditions are two types of assertions. Preconditions are assertions about a program's state when a method is invoked, and postconditions are assertions about a program's state after a method finishes.

Java includes two versions of the *assert* statement for validating assertions programmatically. The assert statement evaluates a boolean expression and determines whether it is true or false. The first form of the assert statement is –

assert expression;

This statement evaluates *expression* and throws an *AssertionError* if the expression is false. The second form is –

assert expression1 : expression2;

This statement evaluates *expression1* and throws an *AssertionError* with *expression2* as the error message if *expression1* is false.

The example in fig. 5.5 demonstrates the functionality of the assert statement. Line 11 prompts the user to enter a number between 0 and 10, then line 12 reads the number from the command line. The assert statement at line 15 determines whether the user entered a number within the valid range. If the number is out of range, then the program reports an error. Otherwise the program proceeds normally.

```
1   //Fig. 5.5; AssertTest.java
2   //Demonstrates the assert statement
3   import java.util.Scanner;
4
5   public class AssertTest
6   {
7       public static void main(String args[ ])
8       {
9           Scanner input = new Scanner(System.in);
10
11          System.out.println("Enter a number between 0 and 10:");
12          int number = input.nextInt( );
13
14          //assert that the absolute value is >=0
15          assert(number >=0 && number <=10): "bad number:" + number;
16
17          System.out.println("You entered %d\n", number);
18      }   //end main
19  }   //end class AssertTest

Enter a number between 0 and 10:5
You entered 5

Enter a number between 0 and 10:50
Exception in thread "main" java.lang.AssertionError: bad number: 50
        at AssertTest.main(AssertTest.java:15)
```

Fig. 5.5 Checking with Assert that a Value is Within Range

By default assertions are disabled when executing a program because they reduce performance and are unnecessary for the program's user. To enable assertions at runtime, use the java command's -ea command-line option. To execute the program in fig. 5.5 with assertions enabled, type

java -ea AssertTest

Assertion Enabling and Disabling – By default, assertions are disabled. You enable them by running the program with the -enableassertions or -ea option –

java -enableassertions MyApp

You can even turn on assertions in specific classes or in entire packages. For example –

java -ea: MyClass -ea : com.mycompany.mylib....MyApp

This command turns on assertions for the class MyClass and all classes in the com.mycompany.mylib package and its subpackages. The option -ea... turns on assertions in all classes of the default package.

You can also disable assertions in certain classes and packages with the -disableassertions or -da option –

java -ea: ....-da:MyClass MyAp

Q.31. What is exception handling ? How is it done ? Write a program in C++ that illustrates the exception handling mechanism of C++.

Or

What is exception handling ? Take any one example to show an exception and how can we handle it ? ***(R.G.P.V., Dec. 2016)***

Ans. **Exception Handling** – Refer to Q.19.

C++ Program – The following program illustrates exception handling mechanism –

```
#include <iostream.h>
using namespace std;
int main( )
{
    int a, b;
    cout <<"Enter values of a and b\n";
    cin>>a;
    cin>>b;
    int x = a – b;
    try
    {
        if(x! = 0)
        {
            cout<<"Result(a/x)="<<a/x<<"\n";
        }
        else        //There is an exception
        {
            throw(x); //Throws int object
        }
    }
    catch(int i) //Catches the exception
    {
        cout<<"Exception caught: x="<<x<<"\n";
    }
    cout<<"END";
    return 0;
}
```


INTRODUCTION OF MULTI-THREADING AND DATA COLLECTIONS

Q.32. Write short note on threads in Java. ***(R.G.P.V., June 2016)***

Or

Explain threads in Java programming. ***(R.G.P.V., June 2015)***

Ans. A thread is similar to a program that has a single flow of control. It has a beginning, a body, and an end, and executes commands sequentially. Every program will have at least one thread as shown in fig. 5.6.

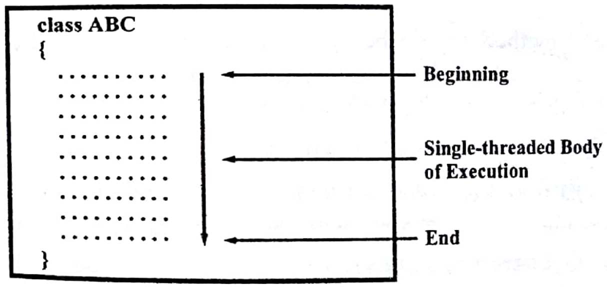


Fig. 5.6 Single-threaded Program

Q.33. Define the thread class.

Ans. Java multithreading system is built upon the *Thread* class, its method and its companion interface called *Runnable*. Thread encapsulates a thread of execution. Since you cannot directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the *Thread* instance that spawned it. For creating a new thread, your program will either extend *Thread* or implement the Runnable interface.

The *Thread* class defines various methods that help manage threads. Some of them are shown in table 5.6 –

Table 5.6

Method	*Task*
getName	Obtain a thread's name
getPriority	Obtain a thread's priority
isAlive	Determine if a thread is still running
join	Wait for a thread to terminate
run	Entry point for the thread
sleep	Suspend a thread for a period of time
start	Start a thread by calling its run method

Q.34. How do we create a thread ?

Ans. Threads are implemented in the form of objects that contain a method called *run()*. The *run()* method is the heart and soul of any thread. It makes up the entire body of a thread and is the method in which the thread's behaviour can be implemented. A typical run( ) would appear as shown below –

```
public void run( )
{
    .................
    ................. (Statements for implementing thread)
    .................
}
```

The *run()* method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called *start()*.

A new thread can be created in two ways –

(i) By Creating a Thread Class – Define a class that extends *Thread* class and override its *run()* method with the code required by the thread.

(ii) By Converting a Class to a Thread – Define a class that implements *Runnable* interface. The *Runnable* interface has only one method, *run()*, that is to be defined in the method with the code to be executed by the thread.

Q.35. In what situation a runnable interface is required to launch a thread?

Ans. For more complicated programs, the preferred way to create a thread is by extending the Runnable interface. The Runnable interface declares the run( ) method that is required for implementing threads in our programs.

Q.36. Explain the difference in syntax and operations among the different methods in Java defined for blocking a thread.

Ans. The following thread methods can be used to temporarily suspend or block a thread from entering into the runnable and subsequently running state–


```
sleep( )      //blocked for a specified time
suspend( )    //blocked until further orders
wait( )       //blocked until certain condition occurs
```

These methods cause the thread to go into the blocked or non-runnable state. The thread will come in the runnable state when the specified time is elapsed in the case of sleep( ), the resume( ) method is called in the case of suspend( ), and the notify( ) method is invoked in the case of wait( ).

Q.37. Explain the lifecycle of a thread in java.

Ans. A thread can enter in many states during it's lifetime. It can move

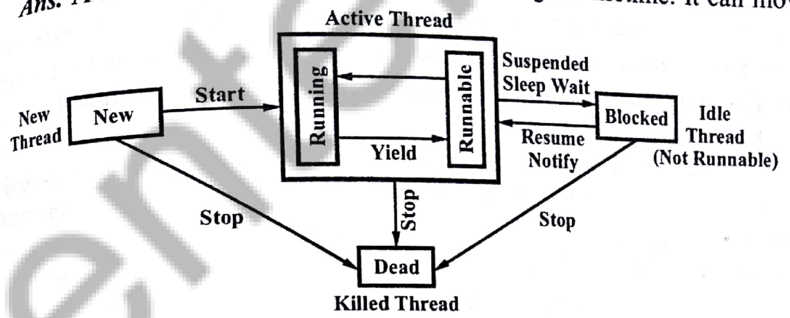


Fig. 5.7 State Transition Diagram of Thread

from one state to another via a variety of ways as shown in fig. 5.7. A thread is always in one of the following states –

(i) New State – A thread enters the newly created state immediately after creation. This thread is created when the new statement is executed. In this state, its local data members are allocated and initialized. The thread is not yet scheduled for running. At this state, we can do only one of the two thing. First, schedule it for running using start( ) method. Second, kill it using stop( ) method. It moves to the runnable state if scheduled. If we attempt to use any other method at this stage an exception will be thrown.

(ii) Runnable State – When a thread is in the runnable state, its ready to execute its 'signature' and can now be in one of two states, running or queued. When a thread is in the running state, it is assigned CPU cycles and is actually running. When a thread is in the queued state, it is waiting in a queue and competing for its turn to send CPU cycles. A runtime scheduler controls the transition between these two substates. However, a thread can call its yield( ) method to voluntarily move itself to the queued state from a running state.

(iii) Blocked State – A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements. A blocked thread is considered not runnable but not dead and therefore fully qualified to run again.

(iv) ***Dead State*** – The dead state is entered when a thread finishes its execution or is stopped by another thread calling its stop( ) method. A thread can be killed as soon it is created, or while it is running, or even when it is in blocked condition.

Q.38. With the help of your own program, explain how threads can be created by –

(i) Extending the Thread class and

(ii) Implementing the Runnable interface.

Ans. (i) Extending the Thread Class – This way to create a new class that extends ***Thread***, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call ***start( )*** to begin execution of the new thread.

Example –

```
//create a second thread by extending thread
class MyThread extends Thread( )
{
    MyThread( )
    {
        super("Demo Thread");
        System.out.println("Child Thread:" + this);
        start( );
    }
public void run( )
{
    try{
        for (int j = 4, j > 0; j --)
        {
            System.out.println("Child Thread:" + j);
            Thread.sleep(500);
        }
    }
        catch(InterruptedException e)
        {
            System.out.println("Child Interrupted");
        }
        System.out.println("Exiting Child Thread");
}
}
class ExtendThread
{
    public static void main(String args[ ])
    {
        //create a new thread
        new MyThread( );
        try
```



```
        {
            for(int j = 4; j > 0; j --)
            {
                System.out.println("Main Thread:" + j)
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e)
        {
            System.out.println("main Thread Interrupted");
        }
        System.out.println("Main Thread Exiting");
    }
}
```

(ii) Implementing the Runnable Interface – The simplest way to create a thread is to create a class that implements the ***Runnable*** interface. Runnable abstracts a unit of executable code. A thread can be constructed on any object that implements ***Runnable***. To implement Runnable, a class need only implement a single method called run( ), which is declared as follows –

public void run( )

Inside ***run( )***, the code that forms the new thread is defined.

After creating a class that implements ***Runnable***, an object of type ***Thread*** from within that class is created. Thread defines several constructors. The one that will be used is as follows –

Thread(Runnable threadOb, string threadName)

In this constructor, *threadOb* is an instance of a class that implements the ***Runnable*** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName.*

After creating the new thread, it will not start running until you call its ***start( )*** method, which is declared within thread. In essence, ***start( )*** executes a call to ***run( )***. The start( ) method is as follows –

void start( );

```
class MyThread implements Runnable
{
    Thread t ;
    MyThread( )
    {
        t = new Thread(this, "Demo Thread");
        System.out.println("Child Thread:" + t);
        t.start( );     //start the thread
    }
    //This is the entry point for the second thread
    public void run( )
```

MSinventer

```
{
        try
        {
            for(j = 4; j > 0; j --)
            {
        System.out.println("Child Thread:" + j);
        Thread.sleep(500);
            }
        }
        catch(InterruptedException e)
        {
            System.out.println("Child Interrupted");
        }
            System.out.println("Exiting child thread");
    }
}
class ThreadDemo
{
    public static void main(String args[ ])
    {
        //creating a new thread
        new MyThread( );
        try
        {
            for(int j = 4; j > 0; j --)
            {
                System.out.println("Main Thread:" + j);
                Thread.sleep(1000);
            }
        }
        catch(InterruptedException e)
        {
            System.out.println("Main Thread Interrupted");
        }
        System.out.println("Main Thread Exiting");
    }
}
```

Q.39. Explain the term multithreading.

Ans. The modern operating systems such as Window 95 and Windows XP can execute several programs simultaneously. This ability is known as ***multitasking***. In system's terminology, it is called ***multithreading***.

In multithreading, a program (process) is divided into two or more subprograms (processes), which can be implemented at the same time in parallel. For example, one subprogram can display an animation on the screen while another may build the next animation to be displayed. This is something

similar to dividing a task into subtasks and assigning them to different people for execution independently and simultaneously.

Multithreading is a unique property of Java. That, is Java enables us to use multiple flows of control in developing program. Each flow of control may be thought of as a separate tiny program (or module) known as thread that runs in parallel to others as shown in fig. 5.8.

A program that contains multiple flows of control is known as ***multithreaded program***.

Fig. 5.8 illustrates a Java program with four threads, one main and three others. The main thread is actually the main method module, which is designed to create and start the other three threads, namely A, B and C. Once initiated by the main thread, the threads A, B and C run concurrently and share the resources jointly. It is like people living in joint families and sharing certain resources among all of them. This ability of a language to support multithreads is referred to as concurrency. Since threads in Java are subprogram of a main application program and share the same memory space, they are known as lightweight threads or lightweight processes.

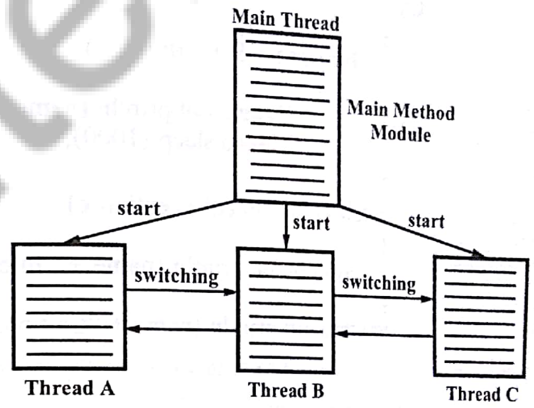


Fig. 5.8 A Multithreaded Program

Q.40. Explain exception handling and multithreading.
(R.G.P.V., Nov. 2018)

Ans. Refer to Q.19 and Q.39.

Q.41. Write a program which contains three threads.

Ans. The following program creates three child threads –

```
// Create multiple threads.
class NewThread implements Runnable
{
    String name;    //name of thread
    Thread t;
    NewThread(String threadname)
```

MSInverter

```
    {
        name = threadname;
        t = new Thread (this, name);
        System.out.println ("New thread:" + t);
        t.start( );      //Start the thread
    }
//  This is the entry point for thread.
    public void run( )
    {
        try
        {
            for (int i = 5; i > 0; i – – )
            {
                System.out.println (name + " : " + i);
                Thread.sleep (1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println (name + "Interrupted");
        }
        System.out.println (name + "Exiting.");
    }
}
class MultiThreadDemo
    {
    public static void main (String args[ ] )
    {
        new NewThread("One");  // start threads
        new NewThread("Two");
        new NewThread("Three");
        try
        {  //wait for other threads to end
           Thread.sleep (10000);
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread Interrupted");
        }
        System.out.println ("Main thread exiting");
        }
    }
```

Output –

```
New thread: Thread [One, 5, main]
New thread: Thread [Two, 5, main]
New thread: Thread [Three, 5, main]
```



```
One : 5
Two : 5
Three : 5
One : 4
Two : 4
Three : 4
One : 3
Three : 3
Two : 3
One : 2
Three : 2
Two : 2
One : 1
Three : 1
Two : 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Q.42. How do we set priorities for threads ? Explain with programme.

Ans. In Java, each thread is assigned a priority, which affect the order in which it is scheduled for running. Thread priorities are integers that specify the relative priority of one thread to another. Theoretically, higher-priority threads get more CPU time than lower-priority threads. But in practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. A higher-priority thread can also preempt a lower-priority one. Thread priorities are used by the thread scheduler to decide when to switch from one running thread to other. This is called a context switch.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems like Windows, threads of equal priority should get equal access to the CPU in round-robin fashion. But you need to be careful. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they do not, the other threads will not run.

Java permit us to set the priority of a thread using the setPriority( ) method, which is a member of *Thread.* Its general form is –

ThreadName.setPriority(intNumber);

Here, intNumber specifies the new priority setting for the calling thread. The value of intNumber must be within the range MIN_PRIORITY and MAX_PRIORITY; these values are 1 and 10, respectively. The default priority of a thread is NORM_PRIORITY whose value is 5.

You can obtain the current priority setting by calling the getPriority( ) method of *Thread* as follows –

```
ThreadName.getPriority( );
class one extends Thread
{
    public void run( )
    {
        System.out.println("Thread one started");
        for(int i = 1; i < = 4; i ++)
        {
            System.out.println("From thread one: i = " +i);
        }
        System.out.println("Exit from thread one");
    }
}
class two extends Thread
{
    public void run( )
    {
    System.out.println("Thread two started");
    for (int j = 1; j < = 4; j++)
    {
    System.out.println("From thread two: j =" +j);
}
    System.out.println("Exit from two");
    }
}
class third extends Thread
{
    public void run( )
    {
        System.out.println("Thread third started");
        for (int k = 1; k < = 4; k ++)
        {
        System.out.println("From thread third : k = " +k);
        }
    System.out.println("Exit from third");
    }
}
class ThreadPriority
{
    public static void main(String args[ ])
    {
        one threadA = new one( );
        two threadB = new two( );
        three threadC = new third( );
        threadC.setPriority(Thread.MAX_PRIORITY);
```



```
        threadB.setPriority(ThreadA.getPriority( ) + 1);
        threadA.setPriority(Thread.MIN_PRIORITY);
        System.out.println("Start thread one");
        threadA.start( );
        System.out.println("Start thread two");
        threadB.start( );
        System.out.println("Start thread third");
        threadC.start( );
        System.out.println("End of main thread");
        }
}
```

Q.43. Explain the term thread synchronization.

Or

Define synchronization. Explain synchronization methods and synchronization block.

Ans. When multiple threads share an object and modified by one or more of the threads, indeterminate results may occur unless access to the shared object is managed properly. If one thread is in the process of updating a shared object and another thread also tries to update it, it is unclear which thread's update takes effect. When this happens, the program's behaviour cannot be trusted. It means that sometimes the program will produce the correct results, and sometimes it will not. In either case, there will be no indication that the shared object was manipulated incorrectly.

The problem can be solved by giving only one thread at a time exclusive access to code that manipulates the shared object. During that time, other threads desiring to manipulate the object are kept waiting. When the thread with exclusive access to the object finishes manipulating it, one of the threads that was waiting is allowed to proceed. This process is called ***thread synchronization***. It coordinates access to shared data by multiple concurrent threads. By synchronizing threads, you can ensure that each thread accessing a shared object excludes all other threads from doing so simultaneously–this is called ***mutual exclusion***.

A common way to perform synchronization is to use Java's built-in *monitors*. Every object has a monitor and a monitor lock. The monitor ensures that its object's monitor lock is held by a maximum of only one thread at any time, and thus can be used to enforce mutual exclusion. If an operation requires the executing thread to hold a lock while the operation is performed, a thread must acquire the lock before it can proceed with the operation. Any other threads attempting to perform an operation that requires the same lock will be *blocked* until the first thread releases the lock, at which point the *blocked* threads may attempt to acquire the lock.

You can synchronize your code in two ways using the *synchronized* keyword.

(i) Using Synchronized Methods – The keyword synchronized helps to solve synchronization problem by keeping a watch on shared resources. For example, the method that will read information from a file and the method that will update the same file may be declared as synchronized. Example –

```
synchronized void update( )
{
    . . . . . . . .
    . . . . . . . .  //code here is synchronized
    . . . . . . . .
}
```

When we declare a method synchronized, Java creates a monitor and hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of code. A monitor is like a key and the thread that holds the key can only open the lock.

(ii) The Synchronized Statement – It is also possible to mark a block of code as synchronized as shown below –

```
synchronized (lock-object)
{
    . . . . . . .  //code here is synchronized
    . . . . . . .
}
```

Whenever a thread has completed its work of using synchronized method (or block of code), it will hand over the monitor to the next thread that is ready to use the same resource.

Q.44. Describe the thread scheduling.

Ans. Every Java thread has a thread priority that helps the operating system determine the order in which threads are scheduled. The purpose of a scheduling policy is to determine how the operating system scheduler manages the thread lists, that is, how threads are moved between and within lists when they gain or lose access to the CPU. Associated with each scheduling policy is a priority range into which all threads scheduled according to that policy must lie. Java priorities range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5). Each new thread inherits the priority of the thread that created it. Informally, higher-priority threads are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute.

Most operating systems support timeslicing, which enables threads of equal priority to share a processor. Without timeslicing, each thread in a set of

equal-priority threads runs to completion before other threads of equal priority get a chance to execute. With timeslicing, even if a thread has not finished executing when its quantum expires, the processor is taken away from the thread and given to the next thread of equal priority, if one is available.

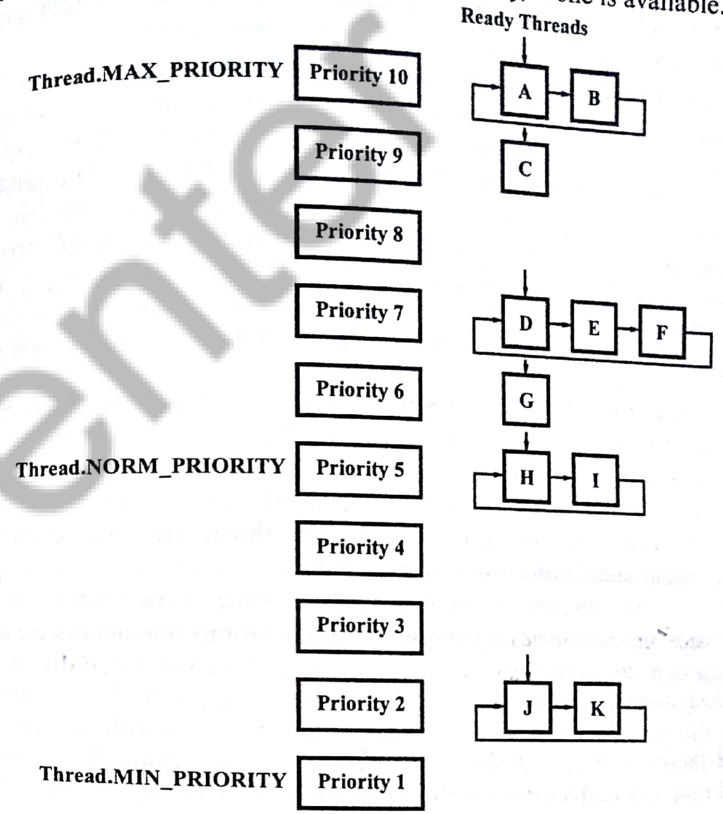


Fig. 5.9 Thread-priority Scheduling

An operating system's thread scheduler determines which thread runs next. One simple thread scheduler implementation keeps the highest-priority thread running at all times and, if there is more than one highest-priority thread, ensures that all such threads execute for a quantum each in round-robin fashion. Fig. 5.9 shows a multilevel priority queue for threads. In the figure, assuming a single processor computer, threads A and B each execute for a quantum in round-robin fashion until both threads complete execution. This means that A gets a quantum of time to run. Then B gets a quantum. Then A gets another quantum. Then B gets another quantum. This continues until one thread completes. The processor then devotes all its power to the thread that remains. Next, thread C runs to completion. Threads D, E and F each execute for a quantum in round-robin fashion until they all complete execution. This process continues until all threads run to completion.

When a higher-priority thread enters the ready state, the operating system generally preempts the currently running thread. Depending on the operating system, higher-priority threads could postpone–possibly indefinitely, the execution of lower-priority threads. Such indefinite postponement is sometimes referred to as starvation.

Thread scheduling is platform dependent, the behaviour of a multithreaded program could vary across different Java implementations.

Q.45. What is the producer/consumer relationship ?

Ans. In a producer/consumer relationship, the producer portion of an application generates data and stores it in a shared object, and the consumer portion of an application reads data from the shared object. The producer/consumer relationship separates the task of identifying work to be done from the tasks involved in actually carrying out the work. An example of a common producer/consumer relationship is print spooling. Another example of the producer/consumer relationship is an application that copies data onto CDs by placing data in a fixed size buffer, which is emptied as the CD-RW drive "burns" the data onto the CD.

In a multithreaded producer/consumer relationship, a producer thread generates data and places it in a shared object called a buffer. A consumer thread reads data from the buffer. This relationship requires synchronization to ensure that values are produced and consumed properly. Operations on the buffer data shared by a producer and consumer thread are state dependent – the operations should proceed only if the buffer is in the correct state. If the buffer is in a not-full state, the producer may produce. If the buffer is in a not empty state, the consumer may consume. All operations that access the buffer must use synchronization to ensure that data is written to the buffer or read from the buffer only if the buffer is in the proper state. If the producer attempting to put the next data into the buffer determines that it is full, the producer thread should wait until there is space to write a new value. If a consumer thread finds the buffer empty or finds that the previous data has already been read, the consumer must also wait for new data to become available.

Q.46. Describe producer/consumer relationship synchronization using the class ArrayBlockingQueue.

Ans. One way to synchronize producer and consumer threads is to use classes from Java's concurrency package that encapsulate the synchronization for you. Java includes the class ArrayBlockingQueue, a fully implemented, thread-safe buffer class that implements interface BlockingQueue. This interface extends the Queue interface and declares methods put and take, the blocking equivalents of Queue methods offer and poll, respectively. Method put places an element at the end of the BlockingQueue, waiting if the queue is full. Method take removes an element from the head of the BlockingQueue, waiting if the queue is empty. These methods make class ArrayBlockingQueue a good choice for implementing a shared buffer. Because method put blocks until there is room in the buffer to write data, and method take blocks until there is new data to read, the producer

must produce a value first, the consumer correctly consumes only after the producer writes a value and the producer correctly produces the next value only after the consumer reads the previous value. ArrayBlockingQueue stores the shared data in an array. The array's size is specified as an argument to the ArrayBlockingQueue constructor. Once created, an ArrayBlockingQueue is fixed in size and will not expand to accommodate extra elements.

The program in fig. 5.10 and fig. 5.11 demonstrates a Producer and a Consumer accessing an ArrayBlockingQueue. Class BlockingBuffer (fig. 5.10) uses an ArrayBlockingQueue object that stores an Integer (line 7). Line 11 creates the ArrayBlockingQueue and passes 1 to the constructor so that the object holds a single value.

```
1   //Fig. 5.10; BlockingBuffer.java
2   //Creates a synchronized buffer using an ArrayBlockingQueue.
3   import java.util.concurrent.ArrayBlockingQueue;
4
5   public class BlockingBuffer implements Buffer
6   {
7       private final ArrayBlockingQueue<Integer> buffer; //shared buffer
8
9       public BlockingBuffer( )
10      {
11          buffer = new ArrayBlockingQueue<Integer>(1);
12      }   //end BlockingBuffer constructor
13
14      //place value into buffer
15      public void set (int value) throws InterruptedException
16      {
17          buffer.put (value); //place value in buffer
18          System.out.printf("%s%2d\t%s%d\n", "Producer writes", value,
19              "Buffer cells occupied:", buffer.size( ));
20      }   //end method set
21
22      //return value from buffer
23      public int get( ) throws InterruptedException
24      {
25          int readValue = 0;  //initialize value read from buffer
26
27          readValue = buffer.take( ); //remove value from buffer
28          System.out.printf("%s %2d\t%s%d\n", "Consumer reads",
29              readValue, "Buffer cells occupied:", buffer.size( ));
30
31      return readValue;
32      }   //end method get
33  }   //end class BlockingBuffer
```

Fig. 5.10 Creates a Synchronized Buffer using an ArrayBlockingQueue

BlockingBuffer implements interface Buffer and use classes Producer and Consumer. This approach demonstrates that the threads accessing the shared object are unaware that their buffer accesses are now synchronized. The synchronization is handled entirely in the set and get methods of BlockingBuffer by calling the synchronized ArrayBlockingQueue methods put

and take, respectively. Thus, the Producer and Consumer Runnables are properly synchronized simply by calling the shared object's set and get methods.

Line 17 in method set (lines 15-20) calls the ArrayBlockingQueue object's put method. This method call blocks if necessary until there is room in the buffer to place the value. Method get (lines 23-32) calls the ArrayBlockingQueue object's take method (line 27). This method call blocks if necessary until there is an element in the buffer to remove. Lines 18-19 and 28-29 use the ArrayBlockingQueue object's size method to display the total number of elements currently in the ArrayBlockingQueue.

Class BlockingBufferTest (fig. 5.11) contains the main method that launches the application. Line 11 creates an ExecutorService, and line 14 creates a BlockingBuffer object and assigns its reference to the Buffer variable sharedLocation. Lines 16-17 execute the Producer and Consumer Runnables. Line 19 calls method shutdown to end the application when the threads finish executing the Producer and Consumer tasks.

Note that while methods put and take of ArrayBlockingQueue are properly synchronized, BlockingBuffer methods set and get (fig. 5.10) are not declared to be synchronized. Thus, the statements performed in method set–the put operation (line 19) and the output (lines 20-21) – are not atomic; nor are the statements in method get–the take operation (line 36) and the output (lines 37-38). So there is no guarantee that each output will occur immediately after the corresponding put or take operation, and the outputs may appear out of order. Even they do, the ArrayBlockingQueue object is properly synchronizing access to the data, as evidenced by the fact that the sum of values read by the consumer is always correct.

```
1   //Fig. 5.11; BlockingBufferTest.java
2   //Two threads manipulating a blocking buffer.
3   import java.util.concurrent.ExecutorService;
4   import java.util.concurrent.Executors;
5
6   public class BlockingBufferTest
7   {
8       public static void main(String[ ] args)
9       {
10          //create new thread pool with two threads
11          ExecutorService application = Executors.newCachedThreadPool( );
12
13          //create BlockingBuffer to store ints
14          Buffer sharedLocation = new BlockingBuffer( );
15
16          application.execute(new Producer(sharedLocation));
17          application.execute(new Consumer(sharedLocation));
18
19          application.shutdown( );
20      }  //end main
21  }  //end class BlockingBufferTest
```

Fig. 5.11 Two Threads Manipulating a Blocking Buffer

Output –

```
Producer writes  1        Buffer cells occupied:  1
Consumer reads   1        Buffer cells occupied:  0
Producer writes  2        Buffer cells occupied:  1
Consumer reads   2        Buffer cells occupied:  0
Producer writes  3        Buffer cells occupied:  1
Consumer reads   3        Buffer cells occupied:  0
Producer writes  4        Buffer cells occupied:  1
Consumer reads   4        Buffer cells occupied:  0
Producer writes  5        Buffer cells occupied:  1
Consumer reads   5        Buffer cells occupied:  0
Producer writes  6        Buffer cells occupied:  1
Consumer reads   6        Buffer cells occupied:  0
Producer writes  7        Buffer cells occupied:  1
Consumer reads   7        Buffer cells occupied:  0
Producer writes  8        Buffer cells occupied:  1
Consumer reads   8        Buffer cells occupied:  0
Producer write   9        Buffer cells occupied:  1
Consumer reads   9        Buffer cells occupied:  0
Producer writes  10       Buffer cells occupied:  1
Producer done producing
Terminating Producer
Consumer reads  10        Buffer cells occupied:  0
Consumer read values totaling 55
Terminating Consumer
```

Q.47. Describe producer/consumer relationship synchronization using the synchronized keyword.

Ans. The first step in synchronizing access to the buffer using synchronized keyword is to implement methods get and set as synchronized methods. This needs that a thread obtain the monitor lock on the Buffer object before attempting to access the buffer data, but it does not solve the state-dependence problem associated with producer/consumer relationships. We must ensure that threads proceed with an operation only if the buffer is in the appropriate state. We need a way to permit our threads to wait, depending on whether certain conditions are true. In the case of placing a new item in the buffer, the condition that allows the operation to proceed is that the buffer is not full. In the case of fetching an item from the buffer, the condition that allows the operation to proceed is that the buffer is not empty. If the condition in question is true, the operation may proceed. If it is false, the thread must wait until it becomes true. When a thread is waiting on a condition, it is removed from contention for the processor, placed in the object's wait queue and the lock it holds is released.

Methods Wait, Notify and NotifyAll – Methods wait, notify and notifyAll, which are declared in class object and inherited by all other classes, are used with conditions to make threads wait when they cannot perform their tasks. If a thread obtains the monitor lock on an object, then determines that it cannot

continue with its task on that object until some condition is satisfied, the thread can call object method wait. This releases the monitor lock on the object, and the thread waits in the *waiting* state while the other threads try to enter the object's synchronized statement(s) or method(s). When a thread executing a synchronized statement (or method) satisfies the condition on which another thread may be waiting, it can call object method notify to allow a waiting thread to transition to the *runnable* state again. At this point, the thread that was transitioned from the *wait* state to the *runnable* state can attempt to reobtain the monitor lock on the object. Even if the thread is able to reobtain the monitor lock, it still might not be able to perform its task at this time–in which case the thread will reenter the *waiting* state and implicitly release the monitor lock. If a thread calls notifyAll, then all the threads waiting for the monitor lock become eligible to reobtain the lock i.e., they all transition to the *runnable* state.

The application in fig. 5.12 and fig. 5.13 shows a Producer and a Consumer accessing a shared buffer with synchronization. In this case, the Producer always produces a value first, the Consumer correctly consumes only after the Producer produces a value and the Producer correctly produces the next value only after the Consumer consumes the previous value. The synchronization is handled in the set and get methods of class SynchronizedBuffer (fig. 5.12), which implements interface Buffer (line 4). Thus, the Producer's and Consumer's run methods call the shared object's synchronized set and get methods.

```
1   //Fig. 5.12; SynchronizedBuffer.java
2   //Synchronizing access to shared data using object
3   //methods wait and notify.
4   public class SynchronizedBuffer implements Buffer
5   {
6       private int buffer = –1;     //shared by producer and consumer threads
7       private boolean occupied=false;     //whether the buffer is occupied
8
9       //place value into buffer
10      public synchronized void set(int value)
11      {
12          //while there are no empty locations, place thread in waiting state
13          while (occupied)
14          {
15              //output thread information and buffer information, then wait
16              System.out.println("Producer tries to write.");
17              displayState("Buffer full. Producer waits");
18              wait( );
19          }   //end while
20
21          buffer = value; //set new buffer value
22
23          //indicate producer cannot store another value
24          //until consumer retrieves current buffer value
25          occupied = true;
26
27          displayState("Producer writes" +buffer);
28
29          notifyAll( ); //tell waiting thread(s) to enter runnable state
30      }   //end method set; releases lock on SynchronizedBuffer
31      //return value from buffer
32      public synchronized int get( )
33      {
34          //while no data to read, place thread in waiting state
35          while (!occupied)
36          {
37              //output thread information and buffer information, then wait
38              System.out.println("Consumer tries to read.");
39              displayState("Buffer empty. Consumer waits.");
40              wait( );
41          }   //end while
42
43          //indicate that producer can store another value
44          //because consumer just retrieved buffer value
45          occupied = false;
46
47          displayState("Consumer reads "+buffer);
48
49          notifyAll( ); //tell waiting thread(s) to enter runnable state
50
51          return buffer;
52
53      }   //end method get; releases lock and SynchronizedBuffer
54
55      //display current operation and buffer state
56      public void displayState(String operation)
57      {
58          System.out.printf("%-40s%d\t\t%b\n\n", operation, buffer,
59              occupied);
60      }   //end method displayState
61  } //end class SynchronizedBuffer
```


Fig. 5.12 Synchronizing Access to Shared Data using Object Methods Wait and Notify

Fields and Methods of Class SynchronizedBuffer – Class SynchronizedBuffer has two fields–buffer (line 6) and occupied (line 7). Method set (lines 10-30) and method get (lines 33-53) are declared as synchronized i.e., only one thread can call either of these methods at a time on a particular SynchronizedBuffer object. Field occupied is used to determine whether it is the Producer's or the Consumer's turn to perform a task. This field is used in conditional expressions in both the set and get methods. If occupied is false, then buffer is empty, so the Consumer cannot read the value of buffer, but the Producer can place a value into buffer. If occupied is true, the Consumer can read a value from buffer, but the Producer cannot place a value into buffer.

Method set and the Producer Thread – When the Producer thread's run method invokes synchronized method set, the thread implicitly attempts to obtain the SynchronizedBuffer object's monitor lock. If the monitor lock is available, the Producer thread implicitly acquires the lock. Then the loop at lines 13-19 first determines whether occupied is true. If so, buffer is full, so line 16 outputs a message indicating that the Producer thread is trying to write a value, and line 17 invokes method displayState (lines 56-60) to output another message indicating that buffer is full and that the Producer thread is waiting

MSinventer

until there is space. Line 18 invokes method wait to place the thread that called method set (i.e., the Producer thread) in the *waiting* state for the SynchronizedBuffer object. The call to wait causes the calling thread to implicitly release the lock on the SynchronizedBuffer object. This is important because the thread cannot currently perform its task and because other thread (in this case, the Consumer) should be allowed to access the object to allow the condition (occupied) to change. Now another thread can attempt to obtain the SynchronizedBuffer object's lock and invoke the object's set or get method.

The Producer thread remains in the *waiting* state until another notifies the Producer that it may proceed–at which point the Producer returns to the *runnable* state and attempts to implicitly reacquire the lock on the SynchronizedBuffer object. If the lock is available, the Producer thread reacquires the lock, and method set continues executing with the next statement after the wait call. Because wait is called in a loop, the loop-continuation condition is tested again to determine whether the thread can proceed. If not, then wait is invoked again. Otherwise, method set continues with the next statement after the loop.

Line 21 in method set assigns the value to the buffer. Line 25 sets occupied to true to indicate that the buffer now contains a value (i.e., a consumer can read the value, but a Producer cannot yet put another value there). Line 27 invokes method displayState to output a message indicating that the Producer is writing a new value into the buffer. Line 29 invokes method notifyAll. If any threads are waiting on the SynchronizedBuffer object's monitor lock, those threads enter the *runnable* state and can now attempt to reacquire the lock. Method notifyAll returns immediately, and method set then returns to the calling method (i.e., the Producer's run method). When method set returns, it implicitly releases the monitor lock on the SynchronizedBuffer object.

Method get and the Consumer Thread – Methods get and set are implemented in similar fashion. When the Consumer thread's run method invokes synchronized method get, the thread attempts to obtain the monitor lock on the SynchronizedBuffer object. If the lock is available, the Consumer thread obtains it. Then the while loop at lines 36-42 determines whether occupied is false. If so, the buffer is empty, so line 39 outputs a message indicating that the Consumer thread is trying to read a value, and line 40 invokes method displayState to output a message indicating that the buffer is empty and that the Consumer thread is waiting. Line 41 invokes method wait to place the thread that called method get (i.e., the Consumer) in the *waiting* state for the SynchronizedBuffer object. Again, the call to wait causes the calling thread to implicitly release the lock on the SynchronizedBuffer object, so another thread can attempt to acquire the SynchronizedBuffer object's lock and invoke the object's set or get method. If the lock on the SynchronizedBuffer is not available (for example, if the Producer has not yet returned from method set), the Consumer is blocked until the lock becomes available.

The Consumer thread remains in the *waiting* state until it is notified by another thread that it may proceed–at which point the Consumer thread returns

to the *runnable* state and attempts to implicitly reacquire the lock on the SynchronizedBuffer object. If the lock is available, the Consumer obtains the lock, and method get continues executing with the next statement after wait. Because wait is called in a loop, the loop-continuation condition is tested again to determine whether the thread can proceed with its execution. If not, wait is invoked again–otherwise, method get continues with the next statement after the loop. Line 46 sets occupied to false to indicate that buffer is now empty (i.e., a consumer cannot read the value, but a Producer can place another value in buffer), line 48 calls method notifyAll. If any threads are in the *waiting* state for the lock on this SynchronizedBuffer object, they enter the *runnable* state and can now attempt to reobtain the lock. Method notifyAll returns immediately, then method get returns the value of buffer to its caller. When method get returns, the lock on the SynchronizedBuffer object is implicitly released.

Testing Class SynchronizedBuffer – Class SharedBufferTest2 (fig. 5.13) contains method main (lines 8-24), which launches the application. Line 11 creates an ExecutorService to run the Producer and Consumer tasks. Line 14 creates a synchronizedBuffer object and assigns its reference to Buffer variable sharedLocation. This object stores the data that will be shared between the Producer and Consumer. Lines 16-17 display the column heads for the output. Lines 20-21 execute a Producer and a Consumer. Finally, line 23 calls method shutdown to end the application when the Producer and Consumer complete their tasks. When method main ends (line 24), the main thread of execution terminates.

```
1  //Fig. 5.13: SharedBufferTest2.java
2  //Two threads manipulating a synchronized buffer.
3  import java.util.concurrent.ExecutorService;
4  import java.util.concurrent.Executors;
5
6  public class SharedBufferTest2
7  {
8      public static void main(String[ ] args)
9      {
10         //create a newCachedThreadPool
11         ExecutorService application = Executors.newCachedThreadPool( );
12
13         //create SynchronizedBuffer to store ints
14         Buffer sharedLocation = new SynchronizedBuffer( );
15
16         System.out.printf("%-40s%s\t\t%s\n%-40s%s\n\n", "Operation",
17              "Buffer", "Occupied", "-------", "------\t\t-------");
18
19
20         //execute the Producer and Consumer tasks
21         application.execute(new Producer(sharedLocation));
22         application.execute(new Consumer(sharedLocation));
23
24         application.shutdown( );
       }   //end main
25 }
   //end class SharedBufferTest2
```

Fig. 5.13 Two Threads Manipulating a Synchronized Buffer

Output –

Operation	*Buffer*	*Occupied*
Consumer tries to read.		
Buffer empty. Consumer waits.	–1	false
Producer writes 1	1	true
Consumer reads 1	1	false
Consumer tries to read.		
Buffer empty. Consumer waits.	1	false
Producer writes 2	2	true
Consumer reads 2	2	false
Producer writes 3	3	true
Consumer reads 3	3	false
Producer writes 4	4	true
Producer tries to write.		
Buffer full. Producer waits.	4	true
Consmer reads 4	4	false
Producer writes 5	5	true
Consumer reads 5	5	false
Producer writes 6	6	true
Producer tries to write.		
Buffer full. Producer waits.	6	true
Consumer read 6	6	false
Producer writes 7	7	true
Producer tries to write.		
Buffer full. Producer waits.	7	true
Consumer reads 7	7	false
Producer writes 8	8	true
Consumer reads 8	8	false
Consumer tries to read.		
Buffer empty. Consumer waits.	8	false
Producer writes 9	9	true
Consumer reads 9	9	false
Consumer tries to read.		
Buffer empty. Consumer waits.	9	false
Producer writes 10	10	true
Consumer reads 10	10	false
Producer done producing		
Terminating Producer		
Consumer read values totaling 55		
Terminating Consumer		

Q.48. Describe producer/consumer relationship synchronization using bounded buffers.

Ans. To reduce the amount of waiting time for threads that share resources and operate at the same average speeds, we can implement a bounded buffer that provides a fixed number of buffer cells into which the Producer can place values, and from which the Consumer can retrieve those values. If the Producer temporarily produces values faster than the Consumer can consume them, the Producer can write additional values into the extra buffer space (if any are available). This capability enables the Producer to perform its task even though the Consumer is not ready to retrieve the current value being produced. Similarly, if the Consumer consumes faster than the Producer produces new values, the Consumer can read additional values from the buffer. This enables the Consumer to keep busy even though the Producer is not ready to produce additional values.

It should be noted that even a bounded buffer is inappropriate if the Producer and the Consumer operate consistently at different speeds. If the Consumer always executes faster than the Producer, then a buffer containing one location is enough. Additional locations would simply waste memory. If the Producer always executes faster, only a buffer with an "infinite" number of locations would be able to absorb the extra production. However, if the Producer and Consumer execute at about the same average speed, a bounded buffer helps to smooth the effects of any occasional speeding up or slowing down in either thread's execution.

The key to using a bounded buffer with a Producer and Consumer that operate at about the same speed is to provide the buffer with enough locations to handle the anticipated "extra" production. If, over a period of time, we determine that the Producer often produces as many as three more values than the Consumer can consume, we can provide a buffer of at least three cells to handle the extra production. Making the buffer too small would cause threads to wait longer. Making the buffer too large would waste memory.

The program in fig. 5.14 and fig. 5.15 demonstrates a Producer and a Consumer accessing a bounded buffer with synchronization. We implement the bounded buffer in class CircularBuffer (fig. 5.14) as a circular buffer that uses a shared array of three elements. A circular buffer writes into and reads from the array elements in order, beginning at the first cell and moving toward the last. When a Producer a Consumer reaches the last element, it returns to the first and begins waiting or reading, respectively, from there. In this version of the producer/consumer relationship, the Consumer consumes a value only when the array is not empty and the Producer produces a value only when the array is not full.

```
//Fig. 5.14: CircularBuffer.java
//Synchronizing access to a shared three-element bounded buffer.
public class CircularBuffer implements Buffer
{

    private final int[ ] buffer = {– 1, – 1, – 1}; //shared buffer

    private int occupiedCells = 0; //count number of buffers used
    private int writeIndex = 0; //index of next element to write to
    private int readIndex = 0; //index of next element to read

    //place .alue into buffer
    public synchronized void set (int value) throws InterruptedException
```

```
13      {
14          //output thread information and buffer information, then wait;
15          //while no empty locations, place thread in blocked state
16          while (occupiedCells = = buffer.length)
17          {
18              System.out.printf("Buffer is full. Producer waits.\n");
19              wait( ); //wait until a buffer cell is free
20          }   //end while
21
22          buffer[writeIndex] = value; //set new buffer value
23
24          //update circular write index
25          writeIndex = (writeIndex + 1) % buffer.length;
26
27          ++occupiedCells; //one more buffer cell is full
28          displayState("Producer writes" + value);
29          notifyAll( ); //notify threads waiting to read from buffer
30      }   //end method set
31
32      //return value from buffer
33      public synchronized int get( ) throws InterruptedException
34      {
35          //wait until buffer has data, then read value;
36          //while no data to read; place thread in waiting state
37          while (occupiedCells = = 0)
38          {
39              System.out.printf("Buffer is empty. Consumer waits.\n");
40              wait( ); //wait until a buffer cell is filled
41          }   //end while
42
43          int readValue = buffer[readIndex]; //read value from buffer
44
45          //update circular read index
46          readIndex = (readIndex +1) % buffer.length;
47
48          --occupiedCells; //one fewer buffer cells are occupied
49          displayState("Consumer reads"+ readValue);
50          notifyAll( ); //notify threads waiting to write to buffer
51
52          return readValue;
53      }   //end method get
54
55      //display current operation and buffer state
56      public void displayState(String operation)
57      {
58          //output operation and number of occupied buffer cells
59          System.out.printf("%s%s%d)\n%s", operation,
60              "(buffer cells occupied:", occupiedCells, "buffer cells:");
61
62          for (int value : buffer)
63              System.out.printf("%2d", value); //output values in buffer
64
65          System.out.print("\n    ");
66
```



```
67          for (int i = 0; i < buffer.length; i++)
68              System.out.print("-----");
69
70          System.out.print("\n    ");
71
72          for (int i = 0; i < buffer.length; i++)
73          {
74              if (i = = writeIndex && i = = readIndex)
75                  System.out.print("WR"); //both write and read index
76              else if (i = = writeIndex)
77                  System.out.print("WR"); //just write index
78              else if (i = = readIndex)
79                  System.out.print("WR"); //just read index
80              else
81                  System.out.print("      "); //neither index
82          }   //end for
83
84          System.out.println("\n");
85      }   //end method displayState
86  }   // end class CircularBuffer
```

Fig. 5.14 Synchronizing Access to a Shared Three-element Bounded Buffer

Line 5 initializes array buffer as a three-element integer array that represents the circular buffer. Variable occupiedCells (line 7) counts the number of elements in buffer that contain data to be read. When occupiedBuffers is 0, there is no data in the circular buffer and the Consumer must wait-when occupied cells is 3 (the size of the circular buffer), the circular buffer is full and the Producer must wait. Variable writeIndex (line 8) indicates the next location in which a value can be placed by a Producer. Variable readIndex (line 9) indicates the position from which the next value can be read by a Consumer.

CircularBuffer method set (lines 12-30) performs the same tasks as in fig. 5.14 with a few modifications. The loop at lines 16-20 determines whether the Producer must wait (i.e., all buffers are full). If so, line 18 indicates that the Producer is waiting to perform its task. Then line 19 invokes method wait, causing the Producer thread to release the CircularBuffer's lock and wait until there is space for a new value to be written into the buffer. When execution continues at line 22 after the while loop, the value written by the Producer is placed in the circular buffer at location writeIndex. Then line 25 updates writeIndex for the next call to CircularBuffer method set. This line is the key to the "circularity" of the buffer. When writeIndex is incremented past the end of the buffer, the line sets it to 0. Line 27 increments occupiedCells, because there is now one more value in the buffer that the Consumer can read. Next, line 28 invokes method displayState (lines 56-85) to update the output with the value produced, the number of occupied buffers, the contents of the buffers and the current writeIndex and readIndex. Line 29 invokes method notifyAll to transition *waiting* threads to the runnable state, so that a waiting Consumer thread (if there is one) can now try again to read a value from the buffer.

The loop at lines 37-41 determines whether the Consumer must wait (i.e., all buffer cells are empty). If the Consumer must wait, line 39 updates the output to indicate that the Consumer is waiting to perform its task. Then line 40 invokes method wait, causing the current thread to release the lock on the CircularBuffer and wait until data is available to read. When execution eventually continues at line 43 after a notifyAll call from the Producer, readValue is assigned the value at location readIndex in the circular buffer. Then line 46 updates readIndex for the next call to CircularBuffer method get. This line and line 25 implement the "circularity" of the buffer. Line 48 decrements occupiedCells, because there is now one more position in the buffer in which the Producer thread can place a value. Line 49 invokes method displayState to update the output with the consumed value, the number of occupied buffers, the contents of the buffers and the current writeIndex and readIndex. Line 50 invokes method notifyAll to allow any Producer threads waiting to write into the CircularBuffer object to attempt to write again. Then line 52 returns the consumed value to the caller.

Method displayState (lines 56-85) outputs the state of the application. Lines 62-63 output the current values of the buffer cells. Line 63 uses method printf with a "%2d" format specifier to pint the contents of each buffer with a leading space if it is a single digit. Lines 70-82 output the current writeIndex and readIndex with the letters W and R, respectively.

Testing Class CircularBuffer – Class CircularBufferTest (fig. 5.15) contains the main method that launches the application. Line 11 creates the ExecutorService, and line 14 creates a CircularBuffer object and assigns its reference to CircularBuffer variable sharedLocation. Line 17 invokes the CircularBuffer's displayState method to show the initial state of the buffer. Lines 20-21 execute the Producer and Consumer tasks. Line 23 calls method shutdown to end the application when the threads complete the Producer and Consumer tasks.

Each time the Producer writes a value or the Consumer reads a value, the program outputs a message indicating the action performed (a read or a write), the contents of buffer, and the location of writeIndex and readIndex. In the output of fig. 5.15, the Producer first writes the value 1. The buffer then contains the value 1 in the first cell and the value – 1 (the default value for output purposes) in the other two cells. The write index is updated to the second cell, while the read index stays at the first cell. Next, the Consumer reads 1. The buffer contains the same values, but the read index has been updated to the second cell. The Consumer then tries to read again, but the buffer is empty and the Consumer is forced to wait.

```
1    //Fig. 5.15: CircularBufferTest.java
2    //Producer and Consumer threads manipulating a circular buffer.
3    import java.util.concurrent.ExecutorService;
4    import java.util.concurrent.Executors;
5
```



```
     public class CircularBufferTest
6    {
7        public static void main(String[ ] args)
8        {
9            //create new thread pool with two threads
10           ExecutorService application = Executors.newCachedThreadPool( );
11
12           //create CircularBuffer to store ints
13           CircularBuffer sharedLocation = new CircularBuffer( );
14
15           //display the initial state of the CircularBuffer
16           sharedLocation.displayState("Initial State");
17
18           //execute the Producer and Consumer tasks
19           application.execute (new Producer(sharedLocation) );
20           application.execute(new Consumer(sharedLocation) );
21
22           application.shutdown( );
23       }   //end main
24       //end class CircularBufferTest
25   }
```

*Fig. 5.15 **Producer and Consumer Threads Manipulating a Circular Buffer***

Output –

```
Initial State (buffer cells occupied: 0)
buffer cells:   – 1    –1    –1
                ----   ----  ----
                WR
Producer writes 1 (buffer cells occupied: 1)
buffer cells:   1      – 1   –1
                ----   ----  ----
                R      W
Consumer reads 1 (buffer cells occupied: 0)
buffer cells :  1      –1    –1
                ----   ----  ----
                       WR
Buffer is empty. Consumer waits.
Producer writes 2 (buffer cells occupied: 1)
buffer cells:   1      2     –1
                ----   ----  ----
                       R     W
Consumer reads 2 (buffer cells occupied: 0)
buffer cells:   1      2     –1
                ----   ----  ----
                             WR
Producer writes 3 (buffer cells occupied: 1)
buffer cells:   1      2     3
                ----   ----  ----
                W            R
Consumer reads 3 (buffer cells occupied: 0)
buffer cells:   1      2     3
                ----   ----  ----
```

```
                WR
Producer writes 4 (buffer cells occupied: 1)
buffer cells:   4       2       3
               ----    ----    ----
                R       W
Producer writes 5 (buffer cells occupied: 2)
buffer cells:   4       5       3
               ----    ----    ----
                R               W
Consumer reads 4 (buffer cells occupied: 1)
buffer cells:   4       5       3
               ----    ----    ----
                        R       W
Producer writes 6 (buffer cells occupied: 2)
buffer cells:   4       5       6
               ----    ----    ----
                W       R
Producer writes 7 (buffer cells occupied: 3)
buffer cells:   7       5       6
               ----    ----    ----
                        WR
Consumer reads 5 (buffer cells occupied: 2)
buffer cells:   7       5       6
               ----    ----    ----
                        W       R
Producer writes 8 (buffer cells occupied: 3)
buffer cells:   7       8       6
               ----    ----    ----
                                WR
Consumer reads 6 (buffer cells occupied: 2)
buffer cells:   7       8       6
               ----    ----    ----
                R               W
Consumer reads 7 (buffer cells occupied: 1)
buffer cells:   7       8       6
               ----    ----    ----
                        R       W
Producer writes 9 (buffer cells occupied: 2)
buffer cells:   7       8       9
               ----    ----    ----
                W       R
```



```
Consumer reads 8 (buffer cells occupied: 1)
buffer cells:   7       8       9
               ----    ----    ----
                W               R
Consumer reads 9 (buffer cells occupied: 0)
buffer cells:   7       8       9
               ----    ----    ----
                WR
Producer writes 10 (buffer cells occupied: 1)
buffer cells:   10      8       9
               ----    ----    ----
                R       W
Producer done producing
Terminating Producer
Consumer reads 10 (buffer cells occupied: 0)
buffer cells:   10      8       9
               ----    ----    ----
                        WR
Consumer read values totaling: 55
Terminating Consumer
```

Q.49. What do you understand by daemon thread ?

Ans. You can turn a thread into a daemon thread by calling –

t.setDaemon(true);

There is nothing daemonic about such a thread. A daemon is simply a thread that has no other role in life than to serve others. Examples are timer threads that send regular "*timer ticks*" to other threads or threads that clean up state cache entries. When only daemon threads remains, the virtual machine exists. There is no point in keeping the program running if all remaining threads are daemons.

Daemon threads are sometimes mistakenly used by beginners who do not want to think about shutdown actions. However, this can be dangerous. A daemon thread should never access a persistent resource such as file or database since it can terminate at any time, even in the middle of an operation.

Q.50. What is data collection ? Explain in brief about its types.

Or

Write a short note on data collections. ***(R.G.P.V., Nov. 2018)***

Ans. Data collection is a process of collecting data from all the related sources to search solutions to the research problem, test the hypothesis and evaluate the outcomes. Data Collection has two parts –

(i) Primary methods of data collection
(ii) Secondary methods of data collection.

(i) Primary Methods of Data Collection – Primary methods of data collection has two parts –

(a) Quantitative (b) Qualitative.

Msinverter

(a) Quantitative Data Collection Methods – It is based in mathematical calculations in different formats. Quantitative data collection method and analysis involve questionnaires with closed-ended questions, methods of correlation and regression mean, mode and median and others. It is cheaper to apply and they can be applied within shorter period of time compared to qualitative techniques. Usually, it is simple to make comparisons of searchings due to high level of standardization.

(b) Qualitative Data Collection Methods – It does not include numbers or mathematical calculations. Qualitative research is closely associated with words, sounds feeling, emotions, colours and other elements which are non-quantifiable. Qualitative studies aim to ensure greater level of depth of understanding and qualitative data collection techniques involve interviews, questionnaires with open-ended questions, focus groups, observations game or role playing, case study etc.

(ii) Secondary Methods of Data Collection – Secondary data is a type of data that has already been published in books, newspapers, magazines, Journals, online portals etc.

Q.51. What is database management system ?

Ans. A database management system (DBMS) is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to as the ***database,*** contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient. Database systems must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Q.52. What is database approach ?

Ans. In the database approach, a single repository of data is maintained that is defined once and then is accessed by various users. A database system is a collection of interrelated files and a set of programs that allow users to access and modify these files. A major purpose of the database system is to provide users with an abstract view of the data. That is, the system hides certain details of how the data are stored are maintained.

Q.53. What are the properties of database approach ?

Ans. A database has the following implicit properties –

(i) A database represents some aspects of the real world, sometimes called the ***miniworld*** or the ***Universe of Discourse*** (UoD). Changes to the miniworld are reflected in the database.

(ii) A database is a logically coherent collection of data with some inherent meaning. A random assortment of data cannot correctly be referred to as a database.

(iii) A database is designed, built and populated with data for a specific purpose. It has an intended group of users and some preconceived applications in which these users are interested.

Q.54. Discuss some commonly used applications of database system.

Ans. Databases are widely used. Here are some representative applications –

(i) Banking – For customer information, accounts, and loans, and banking transactions.

(ii) Airlines – For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner– terminals situated around the world accessed the central database system through phone lines and other data networks.

(iii) Universities – For student information, course registrations, and grades.

(iv) Credit Card Transactions – For purchases on credit cards and generation of monthly statements.

(v) Telecommunication – For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

(vi) Finance – For storing information about holdings, sales and purchases of financial instruments such as stocks and bonds.

(vii) Sales – For customer, product and purchase information.

(viii) Manufacturing – For management of supply chain and for tracking production of items in factories, inventories of items in warehouses/ stores, and orders for items.

(ix) Human Resources – For information about employees, salaries, payroll taxes and benefits, and for generation of paychecks.

CASE STUDY LIKE – ATM, LIBRARY MANAGEMENT SYSTEM

Q.55. Write case study of the ATM system at user level.

Ans. A requirement documents defines the goal of the ATM system and also working –

Requirements Document – A bank install a new ATM system for allowing users to do some basic transactions. One user can have only one account in single bank. ATM provides some facilities like –

(i) Withdraw money (ii) View account balance
(iii) Deposit cash (iv) Change ATM PIN
(v) Transfer cash to other account (vi) Provides cash receipt.

cash types. Develop

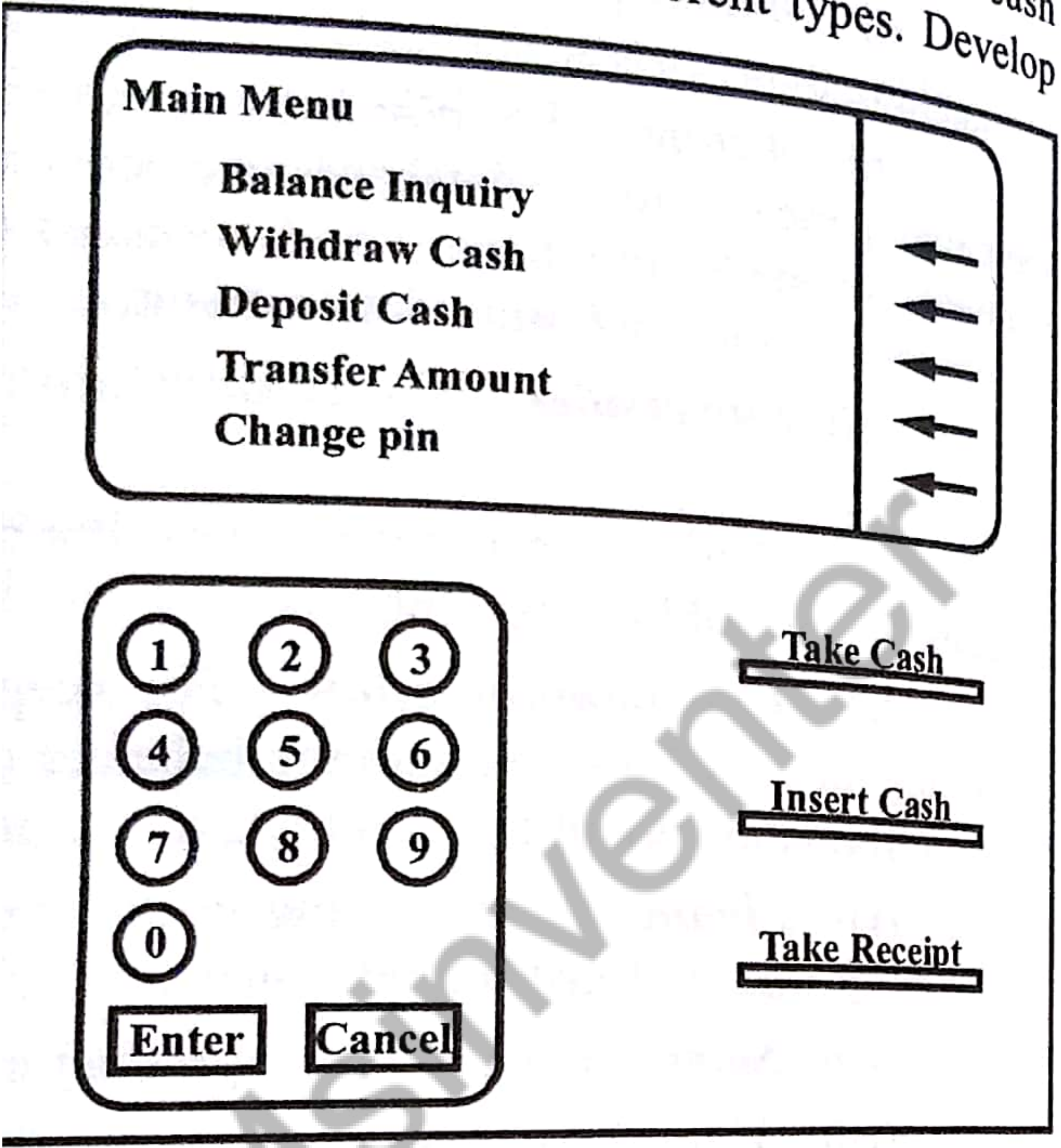


Fig. 5.16 ATM User Interfa

MSInventer

s some Task –

a user via an account number and pin.

ecuting financial transactions.

and Perform Transactions –

nformation databse stores each users account

ATM session interact with bank

```
(10) private static final int Balance_Inquiry = 1;
(11) private static final int Withdrawal =2;
(12) private static final int Deposit = 3;
(13) private static final int Exit = 4;
(14) public ATM ( )
(15) {
(16) UserAuthenticated = false;
```

Class ATM shows the ATM and Line (4) declares an attribute not found in our UML design – an int attribute **Current Account Number** that keeps track of the account number of the current authenticated user.

Lines (5) to (9) declare reference-type attributes belong to the ATM class's associations modeled in fig. 5.17.

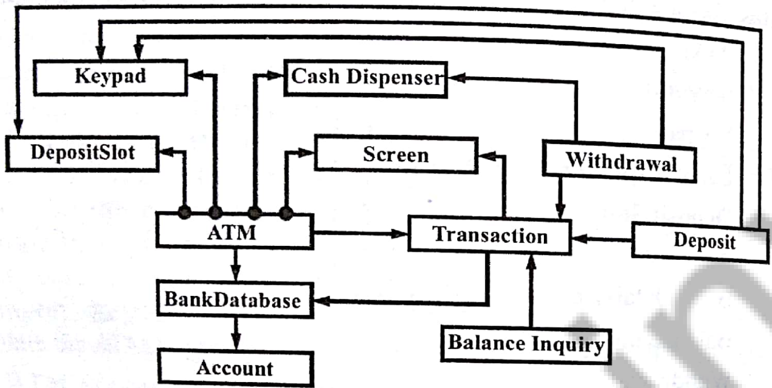


Fig. 5.17

ATM uses these attributes to access its parts and communicate with bank account information database.

```
(17) currentAccountNumber = 0;
(18) screen = new Screen( );
(19) keypad = new Keypad( );
(20) cashDispenser = new CashDispenser( );
(21) depositSlot = new DepositSlot( );
(22) bankDatabase = new BankDatabase( );
(23) }
(24) public void run( )
(25) {
(26)     while (true)
(27)     {
(28)     while (!userAuthenticated)
(29)     {
(30)     screen_displayMessageLine ("Welcome!");
```



```
(31)     authenticateUser( );
(32)     }
(33)     performTransactions( );
(34)     userAuthenticated = false;
(35)     currentAccountNumber = 0;
(36)     Screen.displayMessageLine ("Thank you!")
(37)     }
(38) }
(39) private void authenticateUser( )
(40) {
(41)     screen.displayMessage("Enter your Account number")
(42)         int accountNumber = Keypad.get Input ( );
(43)     screen.displayMessage ("Enter pin");
(44)     int pin = keypad.getInput( );
(45)     UserAuthenticated = bankDatabase.authenticateUser
                                   (accountNumber, pin);
(46)     if (userAuthenticated)
(47)     {
(48)     currentAccountNumber = accoutNumber;
(49)     }
(50)     else
(51)         screen.displayMessageLine("Invalid account number or PIN");
(52)     }
(53)     private void performTransactions( )
(54)     {
(55)     Transaction currentTransaction = null;
(56)     boolean userExited = false;
(57)     while (! userExited)
(58)     {
(59)         int mainMenu Selection = displayMainMenu( );
(60)         switch (mainMenuSelection)
(61)     {
(62)     case Balance_Inquiry :
(63)     case Withdrawal!
(64)     case Deposit :
(65)     currentTransaction =
(66)         createTransaction (mainMenuSelection);
(67)     currentTransaction.execute( );
(68)     break;
(69)     case EXIT :
(70)         screen.displayMessageLine ("Existing the system");
(71)         userExited = true;
(72)         break;
(73)     default;
(74)     screen.displayMessageLine ("you did not enter a valid
                                   section.Try again.");
(75)     break;
```

```
(76)            }
(77)        }
(78)    }
(79)    private int displayMainMenu( )
(80)    {
(81)        screen.displayMessageLine("Main menu");
(82)        screen.displayMessageLine("View account balance");
(83)        screen.displayMessageLine("Withdraw Money");
(84)        screen.dispalyMessageLine("deposite cash");
(85)        screen.displayMessageLine("Exit");
(86)        screen.display MessageLine ("Enter a choice");
(87) return keypad.getInput ( );
(88) }
(89) private Transaction createTransaction(int type)
(90) {
(91)    Transacton temp = null;
(92)    switch (type)
(93)    {
(94)        case Balance_Inquiry :
(95)        temp = new BalanceInquiry (
(96)            current AccountNumber, screen, bank Database);
(97)        break;
(98)    case Withdrawal :
(99)    temp = new Withdrawal(currentAccountNumber;
(100)       screen, bankDatabase, keypad, cashDispenser);
(101)   break;
(102)   case Deposit :
(103)           temp = new Deposit(currentAccountNumber, screen,
(104)           bankDatabase, keypad, depositSlot);
(105)   break;
(106)       }
(107)   return temp;
(108)       }
(109)       }
```

The class diagram does not list any operations for class ATM.

We develop one operation in class ATM for external client of the class to tell the ATM to run.

Run method uses ***infinite loop*** to repeatedly welcome a user, attempt to authenticate the user and if authentications succeeds, user can perform transactions. Simulates the logic which an ATM sees to run continuously as long as bank does not transit off.

An ATM user base not ability to turn off the ATM completely. User can only use the ATM. Lines 28 – 32 cause the ATM to repeatedly welcome and attempt to authenticate the user until the user has been authenticated.

For example, (! UserAuthenticated is true).

Msinvent

Method performTransactions (lines 53 to 78) carries out an ATM session for an authenticated user. We use a **Transaction** variable to take profit of polymorphism.

We name this varaible after the role name involved in the class diagram of fig. 5.18 current trasaction.

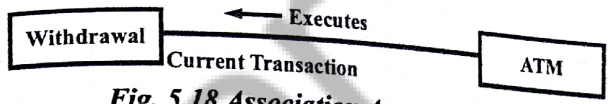


Fig. 5.18 Association Among Classes

Method **creatTransaction** (lines 89 to 108) uses a **switch** statement (lines 92 to 106) to instantiate a new **Transaction** subclass object of the type indicated by the parameter **type.**

Although, if a user selects exit option line 71 sets **userExited** to **true.**

If the user press any other key which is not showing in display option line 74 display an error message, **userExited** remains **false** and the user returns to the main menu.

Q.57. How to return book in library management system ?

Ans. Let us develop a design document for requirement of book return, based on the functional detail of this requirement.

Functional Detail –

Requirement	–	Book return
Input	–	Book ID Number
Output	–	Operation status

Steps –

(i) Check book ID is valid or invalid if invalid perform error routine and stop execution.

(ii) Check that the book is issued to the person who is member of library and will return the book else perform error routine and stop **execution.**

(iii) Ask the librarian if the book is to be returned, if the librarian says no, go to the original book return screen.

(iv) Update the database after perform this operation and inform the librarian of the success else inform about failure with reason.

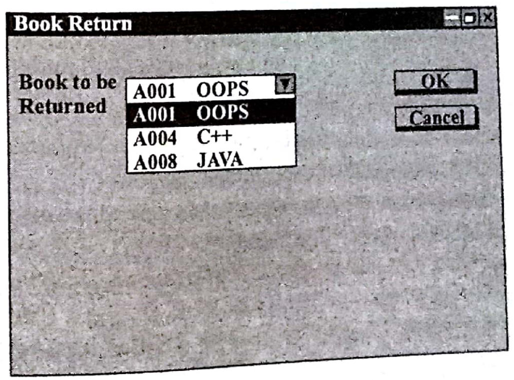


Fig. 5.19

According to requirement let us develop a design

document to address this. The first part of design document show functional requirement.

(i) Functional Requirement – The solution gives the power to the librarian to return back a issue book to user earlier. After this the can issue another book to the member of library. A requirement, which is being manage separately, and is out of the scope of this design.

(ii) User Interface Details – To choose a Book ID number from all book which are available and issue to someone else, the librarian would be displayed a screen.

As display in fig. 5.19, the librarian selects a book ID number. The screen gives him OK and Cancel options.

(a) If the user press Cancel button, application control would return back to the main menu.

(b) If the user press OK button, application would show. The book name, member ID number, and name of person to whom this book was issued and issue date.

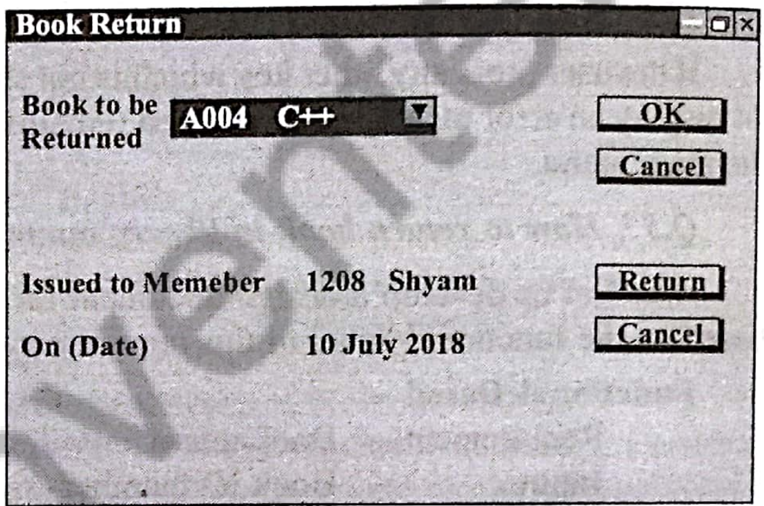


Fig. 5.20

At this juncture, if he wants the books to be returned, the application program asks the librarian.

(1) If librarian press **Cancel** button, control will move to book return dialog box.

(2) If librarian press return button than

(A) The database will update with change the information of the book from issued to available.

(B) The member information will update and now there will not any issue book on his name.

(C) The transaction information will update with book return back date.